

هذا الكتاب
تَرجَمَة لِدَرس Apprenez a programmer en C
الخاص بِمَوقع OpenClassrooms.com

تَرجمة عدن بلواضح
مُراجعَة و إِعداد حمزة عبّاد
تَصميم الغلاف أحمد زيوشي

Mathieu Nebra

تَعَلَّمِ البَرْمَجَة بلُغَة الـC

تَرْجَمَة عدن بلواضح

مُرَاجَعَة و إعْدَاد حمزة عبّاد

تَصميم الغلاف أحمد زبوشي



تنزيل المشروع

تمّت إنشاء هذا الكتّاب بلغة التوصيف LaTeXو ترجمته بمترجم XeLaTeX. يمكن الحصول على الشفرة المصدريّة الخاصة به عن طريق استنساخ مستودع GitHub التالي :

https://github.com/Hamza5/Learn-to-program-with-C_AR

يوجد في هذه الصفحة أيضا رابط لتنزيل النسخة الرقميّة بصيغة PDF، و شرح لطريقة الترجمة و الاعتماديّات الواجب توفّرها، بالإضافة إلى الشفرة المصدريّة الخاصة به.

إذا كنت من مستخدمي GitHub، يمكنك التبليغ عن الأخطاء الّتي قد تجدها في الكتّاب عن طريق فتح issue في هذا المستودع و كتّابة تفاصيل الخطأ (الفصل، القسم، الفقرة، رقم الصفحة و التصحيح الموافق إن أمكن)؛ أو عن طريق القيام بـfork لإنشاء نسخة مطابقة كمستودعك الخاص، ثم إدخال التعديلات المرادة. بعد ذلك، يمكنك القيام بـpull request إلى المستودع الأصلي. في حالة ما كان التعديل جيّدا، سأقوم بدمجه في المستودع.

الترخيص

تقديم

إن التحرّر الفكري في بداية القرن العشرين أدّى إلى توسّع في البحوث العلمية التي شملت كل الميادين لاسيّما التكنولوجية منها كعلوم الحاسوب. هذه الأخيرة أعقبتها ثورة في لغات البرمجة التي تعتبر ركيزة أساسية تقوم عليها البرامج. من بين هذه اللغات نجد لغة الـC، إذ تعتبر من أقوى لغات البرمجة و أكثرها شيوعاً، فهي مستلهمة من طرف لغتي B و BCPL حيث تمّ تطويرها في عام 1972 من طرف Ken Thompson و Dennis Ritchie، و في ظرف سنة واحدة توسّعت لتكون عِماد نظام التشغيل UNIX بنسبة 90% ثم تم توزيعها في العام المُوالي رسمياً عبر الجامعات لتصبح بذلك لغة برمجة عالمية. و اشتهرت لغة الـC كونهُا لغة عالية المستوى، لها مُترجم سريع و فعّال. كما أنها لغة برمجية نقّالة، هذا يعني أن أي برنامج يحترم المعيار AINSI يمكن أن يتمّ تشغيله على أيّة منصّة تحتوي على مترجم C دون أيّة تخصيصات.

يعتبر هذا الكتّاب بوابة سهلة لكلّ مبتدئ لتعلّم لغة الـC خطوة بخطوة بدءً من الأساسيات وصولاً إلى تطوير ألعاب ثنائية الأبعاد و التحكّم في هياكل البيانات الأكثر تعقيداً. الكتّاب مرفق بجملة من التمارين و الأعمال التطبيقية المحلولة التي تساعد على هضم المفاهيم المكتسبة و تطبيقها على أيّ مشكل برمجي مهما كان نوعه. و لأن الكثير من لغات البرمجة تعتمد أساساً على الـC كالـJava و الـC++ و الـC# (لغات برمجية غرضية التوجّه) و حتى PHP (لغة لبرمجة المواقع) فإن تعلّم لغة الـC سيساعد على تعلّم أيّة لغة برمجية كانت. تبقى الإرادة و حبّ العمل و الشغف المفاتيح الرئيسية للنجاح و الوصول إلى الاحترافية.

عدن بلواضح

الجزائر

في 24 ذو القعدة 1438

الموافق لـ17 أوت 2017

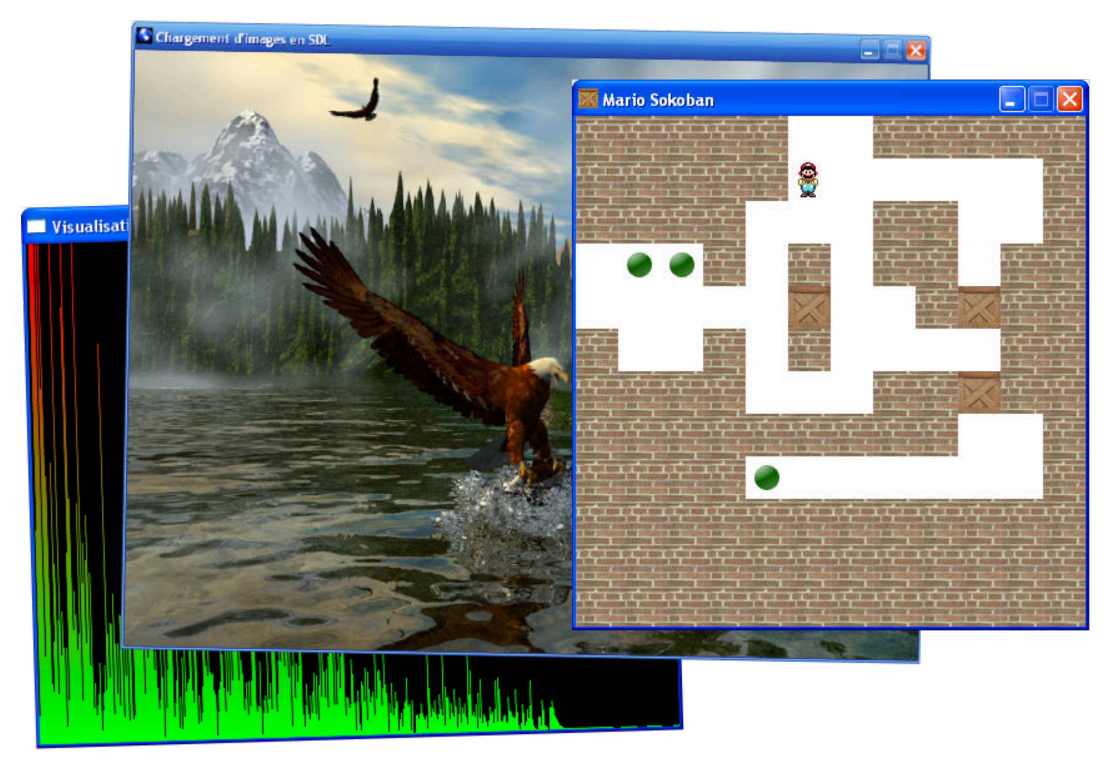
مقدّمة

تحبّ تعلّم البرمجة لكن لا تعرف من أين تبدأ ؟ هذه الدروس لتعليم لغة الـC للمبتدئين قد جُعلت خصّيصاً من أجلك !

لغة الـC هي لغة لا مفرّ منها، أُستلهِمَت منها العديد من اللغات الأخرى. تمّ اختراعها في السبعينات و لا تزال مستعملة لحدّ الآن في البرمجة النظامية و عالم الروبوتات. تعتبر لغة الـC لغة معقّدة، لكن إن استطعت تعلّمها ستكوّن لك قاعدة برمجية صلبة !

في هذه الدروس، ستبدأ باكتشاف مبدأ عمل الذاكرة، المتغيرات، الشروط و الحلقات. ثم ستقوم باستعمال كلّ ما تعلّمته في إنشاء واجهات رسومية بالاستعانة بالمكتبة SDL (ألعاب فيديو، تسجيلات صوتية ...). أخيراً، ستتعلّم كيف تتعامل مع هياكل البيانات الأكثر شيوعاً من أجل تنظيم المعلومات في الذاكرة : قوائم متسلسلة، مكدّسات، طوابير، جداول تجزئة ...

التحق بي في هذه الدروس من أجل اكتشاف البرمجة بلغة الـC !



بعض الإنجازات الّتي سنقوم بها في هذا الكتاب

Mathieu Nebra

مؤسس مشارك لموقع

OpenClassrooms

جدول المحتويات

الجزء ا

أساسيّات البرمجة بلغة الـC

الفصل 1

قلت برمجة ؟

1.1 ما هي البرمجة ؟

؟

ما الذي تعنيه كلمة "بَرْمَجَ"؟

لن أتعبك وأعطيك أصل كلمة "بَرْمَجَ"، لكنني سأختصر كل شيء في جملة : البرمجة تعني إنشاء برامج حاسوب. وهذه البرامج التي تنشئها تأمر الجهاز بالقيام بتعليمات وأفعال معيّنة. حاسوبك الخاص يحتوي على كثير من هذه البرامج وبختلف أنواعها :

- الآلة الحاسبة تعتبر برنامجاً.
- معالج النصوص يعتبر برنامجاً أيضاً.
- وكذلك برنامج المحادثة.
- ألعاب الفيديو هي برامج كذلك.



نسخة عن لعبة MetalSlug الشهيرة تم إنشاؤها من طرف العضو joe87

باختصار البرامج موجودة في كل جهاز، وهي التي تعطي الحاسوب قدرته على إنجاز مختلف المهام التي تُخوَّل إليه. يمكنك أن تنشئ برنامج تشفير أو لعبة ثنائية / ثلاثية الأبعاد باستخدام لغة برمجة مثل C.

ملاحظة : لم أقل أن إنشاء لعبة يتم برمشة عين، لقد قلت فقط بأنه شيء ممكن، لكن كن متأكداً، سوف يتطلب ذلك جهدا كبيراً !

وبما أننا في بداية الطريق، فإننا لن نقوم بإنشاء لعبة ثلاثية الأبعاد ! لكنّنا سنبدأ بكيفية عرض نص على الشاشة، طبعا ستقول ما علاقة هذا بإنشاء الألعاب ؟ لكن ثِق بي، هذا الأمر ليس بسيطا كما يبدو !

بالطبع هذا ليس شيئا مُبهراً، ولكن يجب علينا أن نبدأ من هنا، وشيئا فشيئا يمكنك أن تنشئ برامج معقّدة أكثر. فالهدف من هذا الكتاب هو أن أعرفك على كل ما يتعلق بهذه اللغة.

2.1 البرمجة، بأي لغة يا ترى ؟

حاسوبك هو آلة غريبة جداً، هذا أقل ما يمكن أن نقوله عنه. يمكننا أن نخاطبه فقط بالصفر والواحد، فمثلا إذا طلبنا منه حساب 3+5 فيمكن لهذا أن يعطينا نتيجة كالتالي (هذه ليست ترجمة دقيقة ولكنها تشبه ما يحدث بالفعل):
`0010110110010011010011110`

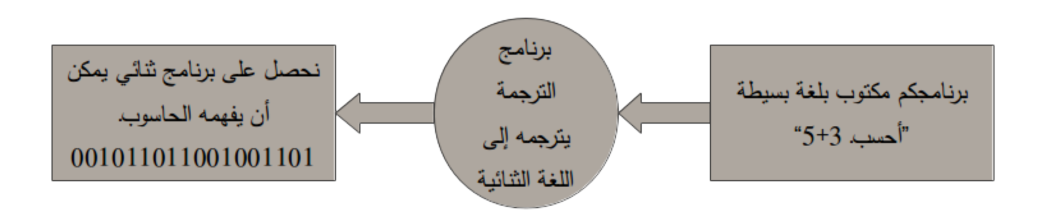
ما تَرَاه هنا يسمى اللغة الثنائية (Binary language) أو لغة الآلة (Machine language)، وحاسوبك لا يفهم سوى هذه اللغة، وكما تلاحظ، هذه اللغة غير مفهومة على الإطلاق !

مشكلتنا الآن :

> ؟ كيف يمكننا التعامل مع حاسوب لا يفهم سوى اللغة الثنائية ؟

حاسوبك لا يتحدث الإنجليزية، ولا العربية، ولا أي لغة غير هذه اللغة، ولكنها صعبة جدا لدرجة أن حتى أكبر خبراء الحاسوب لا يستخدمونها. لهذا قام بعض مهندسي الحواسيب باختراع لغات يمكن أن تُتَرجَمَ إلى اللغة الثنائية، لكن الشيء الأصعب هو إنشاء البرامج الّتي تقوم بهذه الترجمة. ولحسن الحظ فقد قاموا بهذا العمل نيابة عنا. هذه البرامج تقوم بترجمة الأوامر الّتي تكتبها (مثلا : "أُحسب 3+5") إلى شيء يشبه هذا : `0010110110010011010011110`.

هذا المخطط يلخص ما كنت أشرح :



3.1 قليل من المفردات

حتّى الآن كنت أتحدّث إليك بكلمات بسيطة، لكن يجب أن تعلم أنه في المعلوماتية توجد مصطلحات علمية لكل ما ذكرت. طوال هذا الكتاب، سوف نتعلم استخدام المفردات المناسبة. هذا سيفيدك كثيرا خصوصا عندما تحدث مع مبرمجين آخرين، حيث أنك سوف تتفاهم معهم بكل سهولة.

نعود إلى الحديث عن المخطط السابق. في المستطيل الأول قلت أن "برنامجك مكتوب بلغة مُبَسَّطة"، في الواقع هذا النوع من اللغات يُعرف باسم لغات البرمجة عالية المستوى (High-level programming languages). هناك مستويات عديدة من لغات البرمجة، وكلما كان مستوى اللغة أعلى كانت أقرب إلى اللغة الحقيقية وكان استخدامها أسهل. إذن، اللغات عالية المستوى سهلة الاستخدام لكنها تتضمن بعض السلبيّات سوف نتعرّف عليها لاحقا.

توجد العديد من لغات البرمجة، وهي متفاوتة المستوى، منها :

- C
- C++
- Java
- Visual Basic
- Delphi
- و العديد غيرها

كما تلاحظ، لم أرتبها حسب مستوياتها، لذلك لا تعتقد أن اللغة الأولى في القائمة هي الأسهل أو العكس. عموما، لائحة اللغات الموجودة طويلة جدا لدرجة أنه لا يمكنني كتابتها كلها هنا.

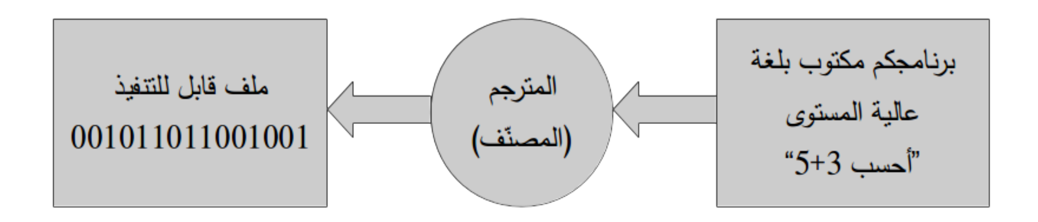
مصطلح آخر يجب تذكّره هو **الشفرة المصدرية** (Source code)، وهي ببساطة الشفرة الخاصة ببرنامجك الذي تكتبه بلغة عالية المستوى والذي يتم ترجمته فيما بعد إلى اللغة الثنائية.

ثم يأتي دور البرنامج الذي يحوّل هذه اللغة عالية المستوى إلى اللغة الثنائية، هذا النوع من البرامج يعرف باسم **المترجم** أو **المصنّف**، والعملية الّتي يقوم بها تسمى **الترجمة** أو **التصنيف**.

> م
> يوجد لكل لغة عالية المستوى مترجم خاص، وهذا شيء منطقي، فاللغات مختلفة فيما بينها، فلا يمكننا ترجمة لغة C بنفس الطريقة الّتي نترجم بها Delphi مثلا. بعض اللغات مثل C تملك العديد من المترجمات، فمنها من هو مكتوب من طرف Microsoft ، و منها من GNU ، إلخ ... سوف نتعرّف على كل هذا في الفصل القادم. لحسن الحظ، هذه المترجمات متطابقة تقريبا (رغم وجود اختلافات طفيفة بينها سوف نتعرف عليها لاحقا).

أخيرا، البرنامج الثنائي المنشئ بواسطة المترجم يسمى الملف **القابل للتنفيذ** أو **التنفيذي** (Executable). لهذا السبب تملك البرامج (على الأقل برامج Windows) الامتداد .exe والذي هو اختصار كلمة EXEcutable.

نعود إلى مخططنا السابق، وهذه المرة سنستخدم المصطلحات الصحيحة :



4.1 لماذا نختار تعلّم C؟

كما قلت سابقا، يوجد كثير من اللغات عالية المستوى، فلماذا ينبغي علينا أن نبدأ بإحداها على وجه الخصوص ؟ سؤال عظيم !

على أية حال يجب علينا أن نختار بأي لغة سنبدأ البرمجة عاجلا أم آجلا، وبالتالي لديك الخيار في البدء بـ :

- **لغة ذات مستوى عالي جدًا** : وتكون سهلة جدًا أوعامة، نذكر من بينها Python، Ruby، Visual Basic، وغيرها. هذه اللغات تسمح بكتابة برامج بشكل أسرع. عامّة تحتاج لأن تُرفق معها ملفات مُسَاعِدة لكي تعمل (كَمُفَسِّرٍ مثلا).
- **لغة ذات مستوى منخفض قليلا** : هي أكثر صعوبة نوعا ما، ولكن مع لغة مثل C سوف تتعلم كثيرا عن البرمجة وحول طريقة عمل حاسوبك. ستكون بعد ذلك قادراً على تعلّم لغة برمجة أخرى إن أردت وبكل يُسْرٍ.

من ناحية أخرى، C لغة برمجة واسعة الإنتشار، أُستخدمت في برمجة العديد من البرامج التي تعرفها. حتى أنها كثيرا ما تدرّس في الدراسات العليا في مجال المعلوماتية. هذه هي الأسباب الّتي جعلتني أتحمّس لتعليمك لغة C بالتحديد. لم أقل أنّه يجب عليك أن تبدأ بها، لكنّي قلت إنه خيار جيّد لكي أقدّم لك معرفة صلبة في هذا الكتاب.

> م
>
> بعض لغات البرمجة موجّهة أكثر للشبكة العنكبوتية (Web) مثل PHP أكثر منها لإنشاء البرامج المعلوماتية.

سوف أفترض في هذا الكتاب أنّ هذه هي لغة برمجتك الأولى وأنّه لم يسبق لك أن برمجت من قبل. فإن كنت قد برمجت قليلا من قبل فلا مضرّة في أن تعيد من الصفر.

> ؟
>
> ما هو الفرق بين C و C++ ؟

هاتان اللغتان قريبتان جدًا من بعضهما، وكلاهما مستخدمتان بكثرة. ولكي تعرف كيف نشأتا يجب عليك أن تدرس التاريخ قليلا :

- في البداية، عندما كانت الحواسيب تَزِنُ أطنانا وتشغل مكانا قَدْرُهُ حجم منزلك، تمّ اختراع لغة برمجة تسمّى Algol.
- بعدها تطوّرت الأمور أكثر واختُرِعَت لغة برمجة جديدة عُرِفَتْ باسْمِ CPL والّتي تطوّرت فيما بعد إلى لغة BCPL ثم أخذت إسم اللغة B.
- مع مضيّ الزمن توصّل الخبراء إلى ابتكار اللغة C وقد تمّ إدخال بعض التعديلات عليها إلّا أنها لا تزال من أحد اللغات الأكثر استخداما اليوم.
- وبعد زمن، أراد الخبراء أن يضيفوا بعض الأشياء إلى C ، يمكن اعتبارها نوعا من التحسينات. والنتيجة كانت بما يعرف بلغة C++ ، وهي لغة C مع إضافات تمكّننا من البرمجة بطريقة مختلفة.

الـ++C ليست أحسن من الـC ، هي فقط تمكننا من البرمجة بطريقة مختلفة وتساعد المبرمج على تنظيم شفرة برنامجه.
رغم ذلك هي تشبه الـC كثيرا. وإن كنت تنوي تعلّم الـ++C فيما بعد فَسَوْفَ تجد ذلك سهلا.

ولو اعتُبِرت ++C تطويرا لـC فإن هذا لا يعني أنه يجب استخدام ++C فقط لإنشاء البرامج. لغة C ليست لغة عجوزا منسيّة، بالعكس هي مستخدمة بكثرة اليوم. بل إنها أساس أنظمة التشغيل الكبيرة مثل Unix (ومنه GNU/Linux و Mac OS) و Windows.

5.1 هل البرمجة صعبة ؟

هذا سؤال يعذّب روح كل من يريد تعلّم البرمجة ! هل يجب أن تكون أستاذ رياضيات كبير درس 10 سنوات من التعليم العالي حتّى تبدأ البرمجة ؟

الجواب هو لا بالطبع. كل ما تحتاج إليه هو معرفة العمليات الأربع الأساسية :

- الجمع
- الطرح
- الضرب
- القسمة

هذا ليس مخيفا ! سوف أشرح لك في فصل لاحق كيف يقوم الحاسوب بهذه العمليات الأساسية في برامجك.

باختصار، لا توجد صعوبات غير قابلة للحلّ. في الواقع، هذا يعتمد على طبيعة برنامجك، فإذا كنت تريد إنشاء برنامج تشفير فيجب عليك معرفة بعض الأشياء في الرياضيات، وإن كان برنامجك يقوم بالرسم ثلاثي الأبعاد فيجب أن تكون لديك بعض المعرفة بالهندسة الفضائية.

كل حالة تعامل بطريقة خاصّة. ولكن لتعلّم لغة C نفسها لا تحتاج إلى أيّة معارف قبليّة.

إذن أين هو الفخ ؟ وأين تكمن الصعوبة ؟

يجب أن تعرف كيف يعمل الحاسوب، لتفهم ما الّذي نقوم به في C. من هذا المنطلق، كن متيقّنا أنّي سأعلّمك كلّ هذا شيئا فشيئا.

اعلم أن للمبرمج صفات أيضا مثل :

- الصبر : البرنامج لا يعمل عادة من أوّل محاولة، يجب أن تكون مثابراً.
- حسّ المنطق : صحيح أنّك لست بحاجة إلى أن يكون لديك مستوى جيّد في الرياضيّات، لكنّ هذا لا يمنع من التفكير وتحليل المشكلات بالمنطق.
- الهدوء : فيجب عليك ألّا تضرب حاسوبك بالمطرقة، فهذا لن يجعل برنامجك يعمل !

الفصل 2

الحصول على الأدوات اللازمة

بعد تجاوزنا لفصل تمهيدي مليئ بالثرثرة سوف نبدأ بالدخول في صلب الموضوع. سوف نجيب عن السؤال التالي : "ما هي البرامج التي نحتاج إليها للبدء في البرمجة ؟".

لا يوجد شيء صعب في هذا الفصل، سوف نأخذ وقتنا للتأقلم على هذه البرامج الجديدة.

اغتنم الفرصة ! في الفصل التالي سنبدأ حقّا في البرمجة و لن يكون هناك وقت للقيلولة !

1.2 الأدوات اللازمة للمبرمج

إذن ما هي الأدوات التي نحتاج إليها ؟ إذا تابعت الفصل السابق جيّدا، فستعرف واحدا على الأقل !

هل تعلم عمّا أتحدّث ؟ حقّا لا ؟

حسنا، نحن نتحدّث عن **المترجم** الذي يمكّن من ترجمة لغة الـC إلى اللغة الثنائيّة !

كما قلت لك في الفصل الأوّل، يوجد العديد من المترجمات للغة الـC. سنرى أن اختيار المترجم ليس أمرا معقّدا في حالتنا هذه.

ما الذي نحتاج إليه أيضا ؟ لن أتركك تخمّن كثيرا و سأعطيك القائمة :

- **محرّر نصوص** (Text Editor) لكتابة الشفرة المصدرية الخاصّة بالبرنامج. نظريّا برنامج تحرير نصوص بسيط مثل Notepad على Windows أو vi على Unix يكفي، لكن من الأحسن استخدام محرّر نصوص ذكيّ يقوم بتلوين الشفرة المصدرية لكي يسهّل عليك العمل.
- **مترجم** لتحويل الشفرة المصدرية إلى ملف ثنائي.
- **المنقّح** (Debugger) لمساعدك على كشف الأخطاء في برنامجك. لسوء الحظ، لم نتمكّن بعد من ابتكار "المصحّح" الذي يصحّح أخطائك لوحده. لكن، إن أحسنت استخدام المنقّح، يمكنك ببساطة إيجاد الأخطاء.

وجود مكتشف الأخطاء لا يعني أن نتصرف بتهوّر و تسرع في كتابة برنامج مليء بالأخطاء، بل تريّث و كن هادئاً.

من الآن لدينا خياران :

- إمّا أن نحصل على البرامج الثلاثة متفرّقة و هذه هي الطريقة الأكثر تعقيدا، و لكنّها تعمل. على GNU/Linux تحديدا، عدد كبير من المبرمجين يفضّلون استخدام كلّ برنامج على حدة. لن أشرح هذه الطريقة هنا، بل سأتحدّث عن الطريقة الأسهل.
- أو أن تحصل على برنامج "ثلاثة في واحد" يتضمّن محرّر النصوص و المترجم و المنقّح. هذا النوع من البرامج يعرف باسم "بيئات التطوير المتكاملة" (Integrated Development Environments) و تسمّى اختصارا IDE.

يوجد العديد من بيئات التطوير. بداية، قد تواجه صعوبة في اختيار البيئة الملائمة لك. الشيء الأكيد هو : أي بيئة مهما كانت ستحقق لك العمل المطلوب.

اختيار البيئة الخاصة بك

بدا لي أنه من الأفضل أن أريك بعضا من البيئات الشهيرة و المجانيّة في نفس الوقت. شخصيّا، أنا أستخدمها جميعا و أختار في كل يوم واحدا منها.

- أحد هذه البيئات الّتي أفضّلها هو **Code::Blocks**. هو مجّاني و يعمل على أغلب أنظمة التشغيل. أنصح كلّ مبتدئ أن يختاره للبدء (و في ما بعد أيضا إذا شعرت أنّه يلائمك جيّدا !).

  يعمل على أنظمة التشغيل Windows، Mac OS و GNU/Linux.
- الأكثر شهرة على Windows هو الّذي أنشأته Microsoft، إنّه **Visual C++**. هو برنامج مدفوع (و باهظ الثمن) لكن لحسن الحظّ توجد نسخة مجانية منه تسمّى **Visual Studio Express** (أنا أستخدم النسخة القديمة **Visual C++ Express** في هذا الكتاب). و هي ممتازة جدّا (بينها و بين النسخة المدفوعة فوارق طفيفة). إنه برنامج كامل و يملك منقّحا قويّا.

  يعمل على Windows فقط.
- على Mac OS X يمكنك استخدام **Xcode** الّذي يفترض أن يكون متوفّرا على قرص تثبيت النظام. يناسب كثيرا مبرمجي Mac.

  يعمل على Mac OS X فقط.

م

ملاحظة لمستخدمي GNU/Linux : يوجد العديد من البيئات لهذا النظام، و لكن المبرمجين المحترفين قد يفضّلون تجاوز البيئات و القيام بالترجمة "يدويّا"، و هو شيء أصعب قليلا. نحن سنبدأ باستخدام بيئات تطويرية. لذلك أنصحك بثبيت Code::Blocks إن كنت على GNU/Linux لكي تتمكن من متابعة شروحاتي.

؟

من هي البيئة الأفضل من بين كلّ بيئات التطوير هذه؟

كل واحدة من هذه البيئات تمكنك من البرمجة و متابعة بقيّة الكتاب من دون أيّة مشاكل. بعضها كامل أكثر من ناحية المميزات، و أخرى سهلة الإستخدام أكثر، ولكن في كلّ الأحوال البرامج الّتي تنشؤها تكون ذاتها أيّا كانت البيئة التي اِختَرْتَها. فهذا الخيار ليس بالأهمّية الّتي تعتقدها.

في هذا الكتاب سوف أستخدم Code::Blocks. فإن أردت الحصول على نفس لقطات الشاشة خاصّتي، خصوصا لكي لا تضيع في البداية، أنصحك بِدَايَةً بتثبيت Code::Blocks.

2.2 Code::Blocks (Windows, Mac OS X, GNU/Linux)

Code::Blocks هي بيئة تطوير متكاملة حرّة و مجانيّة، متوفّرة للـWindows، Mac و GNU/Linux.

حاليّا Code::Blocks متوفّر بالإنجليزيّة فقط. لكن هذا ليس أمرا يدعوك إلى تجنّب استخدامه ! فنحن قلّما نحتاج إلى العمل بقوائم واجهته، فلغة C هي الّتي تهمّنا.

كن على علم أنه عندما تبرمج سوف تقابل عادة توثيقا بالإنجليزية. هذا سبب آخر يدفعك للتدرّب على استخدام هذه اللغة.

تنزيل Code::Blocks

توجّه إلى صفحة تنزيل Code::Blocks

http://www.codeblocks.org/downloads/binaries

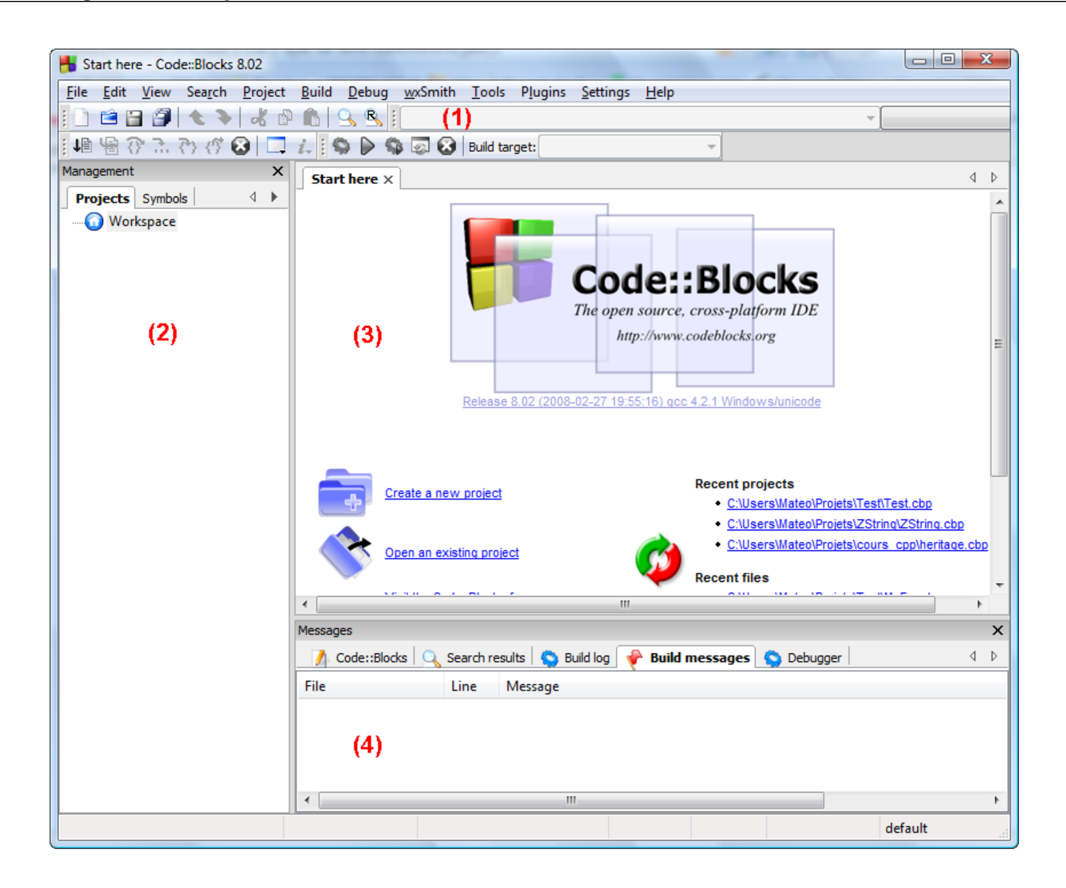
ثمّ نزّل الملف الذي يناسب نظامك :

- إذا كنت تستخدم Windows، اذهب إلى القسم ”Windows“ في أسفل الصفحة. نزّل البرنامج الّذي يحوي `mingw` في اسمه (مثلا : `codeblocks-10.05mingw-setup.exe`). النسخة الأخرى لا تحوي مترجما، لن تتمكن في حال استخدمتها من ترجمة برامجك !
- إذا كنت تستخدم GNU/Linux، اختر الحزمة الّتي تناسب توزيعتك.
- إذا كنت تستخدم Mac، اختر الملف الأحدث في القائمة، مثلا : `codeblocks-10.05-p2-mac.zip`.

X

أقول و أكرّر : إذا كنت تستخدم Windows فيجب عليك تنزيل النسخة الّتي يتضمَن اسمها كلمة mingw لأنّه إذا اخترت النسخة الخاطئة فلن تتمكّن من ترجمة برامجك فيما بعد !

التثبيت بسيط و سريع. أترك جميع الخيارات كما هي و شغّل البرنامج. سوف تظهر لك نافذة شبيهة بهذه :



نميّز أربعة أقسام رئيسية في واجهة البرنامج، و هي مرقّمة في الصورة :

1. شريط الأدوات (Toolbar) : يحتوي على كثير من الأزرار و لكنّنا سوف نستخدم بعضها فقط باستمرار، سأعود لحديث عن هذا فيما بعد.

2. قائمة ملفات المشروع : توجد بيسار النافذة، تحتوي على كلّ الملفات المصدريّة المتعلقة بالبرنامج الذي تعمل عليه. تكون فارغة في البداية لأننا لم ننشئ أي ملف لحد الآن. سوف نبدأ بملأها خلال خمس دقائق من الآن بتقدّمك في هذا الفصل.

3. المنطقة الرئيسية : هنا المساحة التي تكتب فيها الشفرة المصدرية الخاصة ببرنامجك بلغة الـC.

4. منطقة الإشعار : و يدعوها البعض "منطقة الموت"، هنا تُعْرَضُ أخطاء الترجمة إذا كانت شفرة البرنامج تحوي خطأً ما. هذا الشيء يحدث كثيرا !

ما يهمّنا الآن هو قسم محدد من شريط الأدوات. تجد فيه الأزرار التالية (بهذا الترتيب) : `Compile`، `Execute`، `Compile & Execute`، `Recompile everything` تذكّرهم جيدّا لأننا سنستخدمهم بانتظام.



و هذا شرح عمل كلّ واحد من هذه الأزرار :

- **Compile** : كل ملفّات الشفرة المصدرية الّتي كتبتها يتم ارسالها إلى المترجم الّذي يقوم بإنشاء الملف التنفيذي في حالة عدم وجود أخطاء. أمّا في حالة العثور على أخطاء (و هذا سيحدث عاجلا أم آجلا !) فلن يتمّ إنشاؤه بل يعرض رسائل خطأ في منطقة الإشعار.

- **Execute** : يقوم بتشغيل آخر ملف تنفيذي تمّت ترجمته. يعني أنك ستستخدم هذا الزر لاختبار برامجك الّتي أنشأتها. طبعا يجب عليك ترجمة البرنامج قبل تشغيله. يمكننا أيضا استخدام الزرّ الثالث.

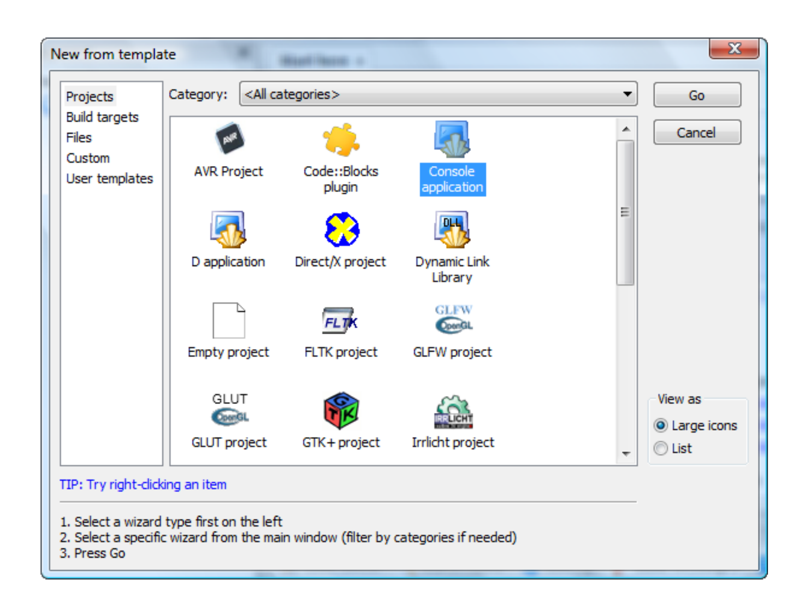
- **Compile & Execute** : لا يجب أن تكون عبقريا لكي تعرف أنّه ليس إلا مجموع الزرّين السابقين. في الواقع هذا هو الزرّ الّذي ستستخدمه أكثر. في حالة ما إذا حدثت أخطاء في الترجمة لن يتمّ تشغيل البرنامج بل ستُعرض قائمة جميلة من الأخطاء التي يجب عليك تصحيحها أوّلا !

- **Recompile everything** : عندما نستخدم **Compile** يقوم Code::Blocks في الحقيقة بترجمة الملفات الّتي عدّلتها فقط. أحيانا -فقط أحيانا- قد تحتاج إلى إعادة ترجمة جميع الملفّات. سنتحدّث لاحقا عن فائدة هذا الزر و أيضا عن كيفية عمل الترجمة بمزيد من التفصيل. حاليا لكي لا تختلط الأمور عليك يمكنك أن تعتبر أنّه غير مهم.

> أنصحك باستخدام اختصارات لوحة المفاتيح بدلا من الضغط على الأزرار، لأنّه شيء نكرّره كثيرا. تذكّر خصوصا أنّه يمكنك الضغط على `F9` بدل الزر `Compile & Execute`.

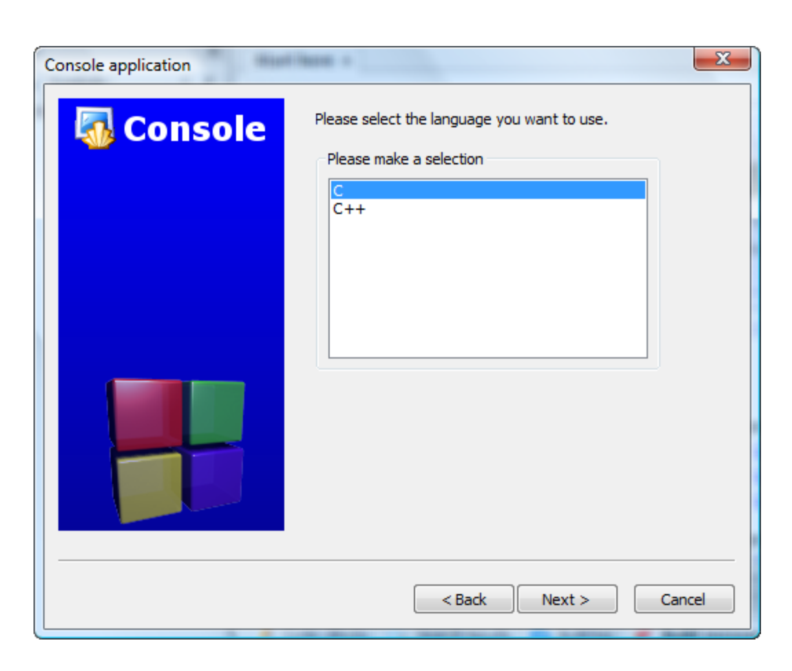
إنشاء مشروع جديد

لإنشاء مشروع جديد إذهب إلى قائمة `File` ثمّ `New` ثمّ `Project`. من النافذة الّتي تظهر اختر `Console application`.

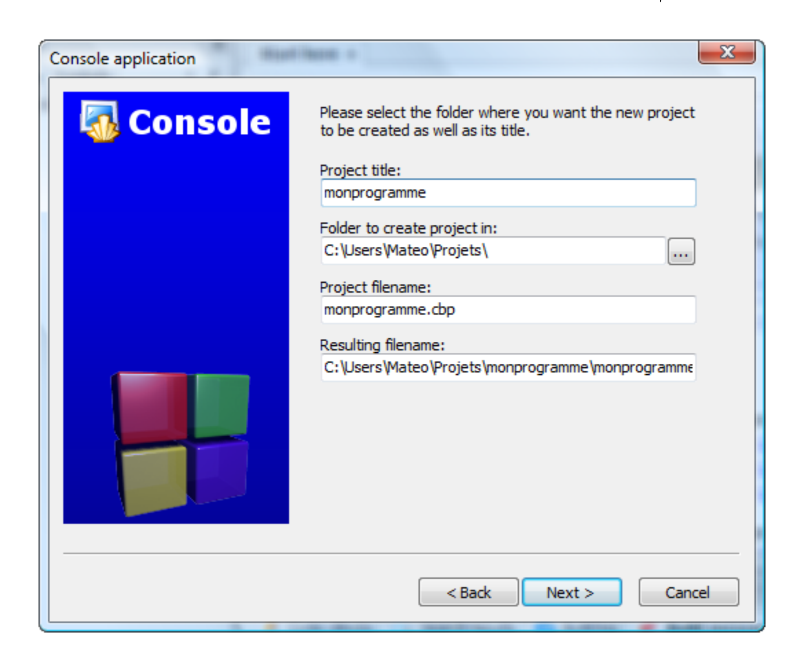


كما ترَى، Code::Blocks يقترح عليك إنشاء عدد معتبر من أنواع البرامج الّتي تستخدم مكتبات (Libraries) معروفة مثل SDL للـ2D و OpenGL للـ3D و Qt و wxWidgets لإنشاء النوافذ الرسوميّة. حاليّا، هذه الأيقونات ليست سوى للزينة لأنّ المكتبات السابقة غير مثبّتة على حاسوبك لهذا لا يمكنك أن تجعلها تعمل. سوف نعود لهذه الأنواع الأخرى لاحقا. في هذه الأثناء لا يمكننا سوى أن نستخدم الـConsole لأنّك لا تملك بعد المستوى اللازم لانشاء أنواع أخرى من البرامج.

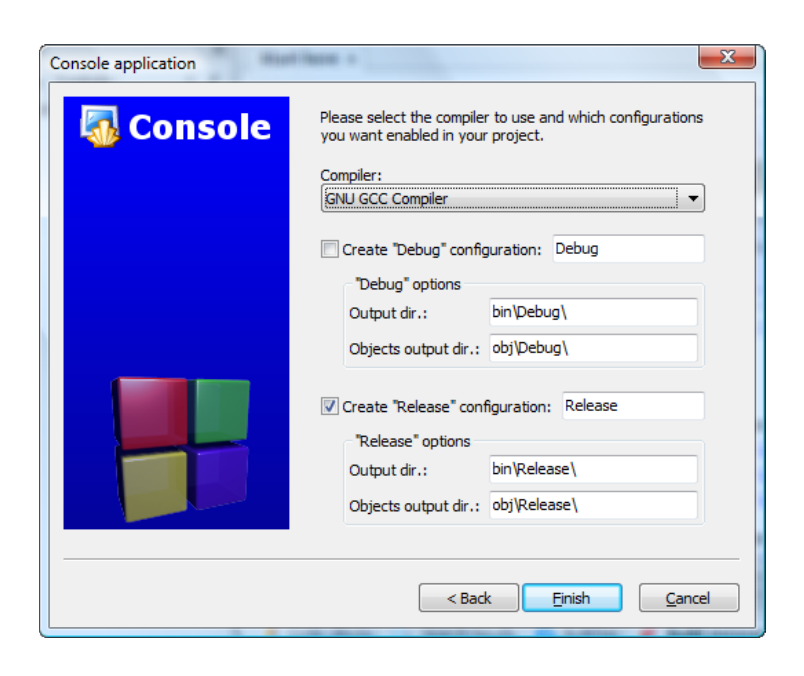
أنقر على `Go` لانشاء المشروع الجديد. ثم أنقر على `Next` فالصفحة الأولى ليس مهمّة. بعدها سيأتيك اختيار بين لغتي الـC أو الـC++، اختر الـC.



سيُطْلَبُ منك الآن إدخال اسم المشروع، و كذلك مسار المجلّد الذي تختاره لحفظ الملفّات فيه.



آخر خطوة تُطلب منك هي ، كيف ينبغي أن يترجم البرنامج، يمكنك ترك الخيارات على حالها، لن يكون لهذا أي تأثير على ما سنقوم به الآن (تأكّد أن إحدى الخانتين "Release" أو "Debug" تكون محدّدة على الأقل).



إضغط على `Finish` ، إنتهى !
لقد قام Code::Blocks بإنشاء المشروع الأوّل و ملئه ببعض الشفرة المصدرية.

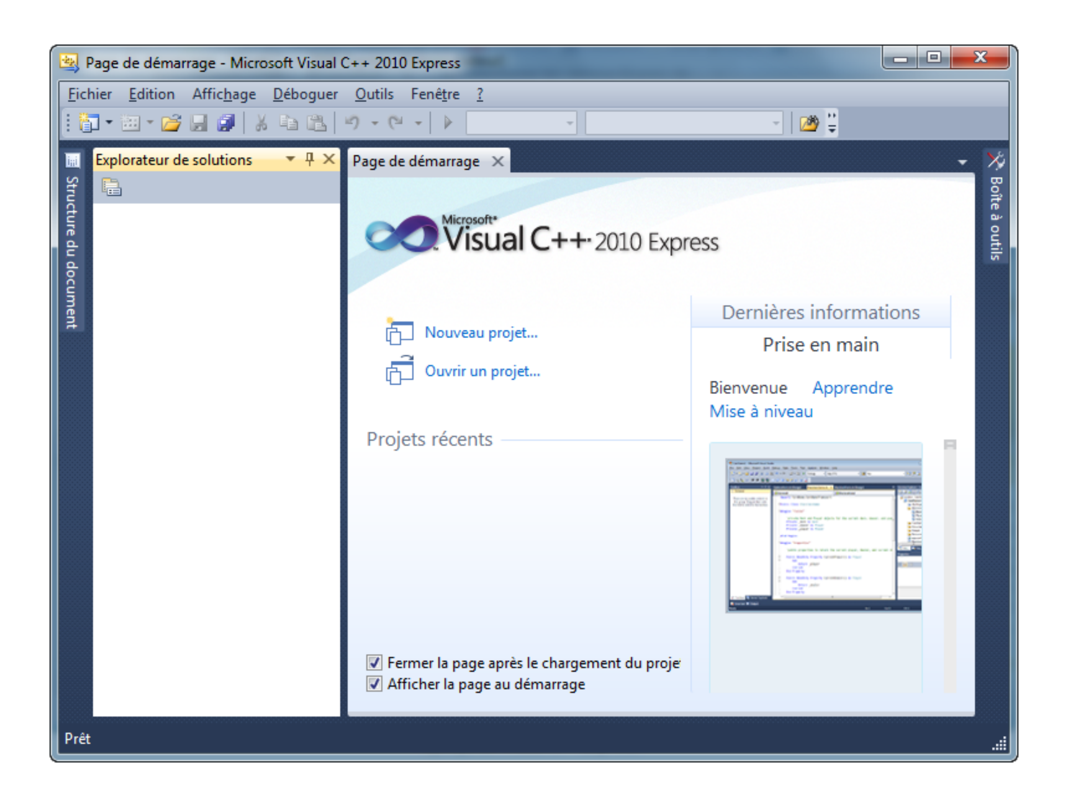
في الخانة الخاصة بالمشاريع على اليسار، قم بتوسيعها بالضغط على '`+`' لكي تظهر قائمة الملفات في المشروع. سيكون لديك على الأقل ملف يسمّى `main.c` . هذا هو كلّ شيء !

3.2 Visual C++ (Windows فقط)

بعض التذكيرات حول Visual C++ :

- إنها البيئة التطويرية الخاصة بـMicrosoft.
- برنامج مدفوع في الأصل، لكن توجد نسخة مجّانية منه تسمّى Visual C++ Express.
- تمكّن من البرمجة باستخدام كلتا اللغتين C و C++ (و ليس فقط C++ كما يوحي الاسم).

طبعا ستقوم بتحميل النسخة المجانية Visual C++ Express (احذر، هو غير متوافق مع Windows 7 إلّا بداية من النسخة 2010) :



ما الفرق بين هذه النسخة و النسخة "الحقيقيّة" ؟

لا تحتوي على محرّر موارد يسمح لك برسم الصور، الأيقونات أو النوافذ. هذا لا يهمّنا لأنّنا لن نحتاج إلى هذه الوظائف في هذا الكتاب. وجود هذه الوظائف أمر مستحسن لكنّه ليس لازما.

للتنزيل، زر موقع Visual C++.

https://msdn.microsoft.com/fr-fr/express/aa975050.aspx

و اختر تنزيل Community 2015 و اختر لغتك المفضّلة.

التثبيت

التثبيت سهل. سوف يقوم البرنامج بتحميل آخر نسخة من الأنترنت تلقائيا.
أنصحك بترك الخيارات كما هي.

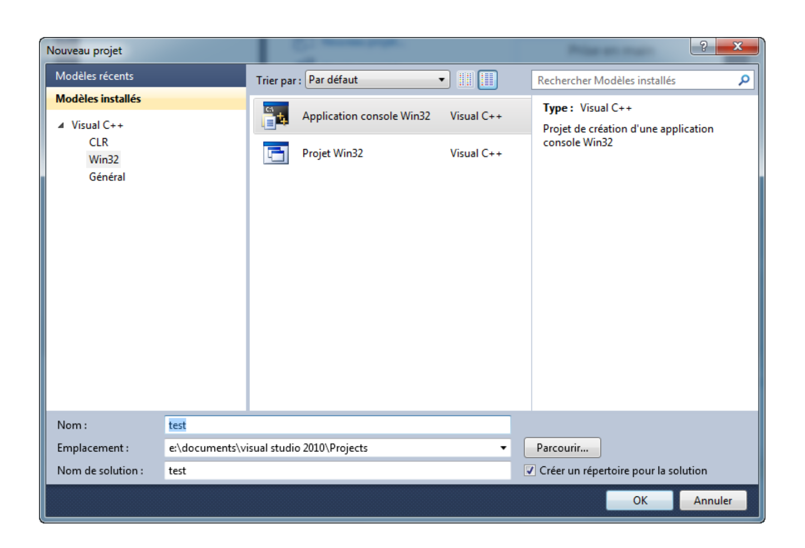
بعد ذلك سيطلب منك التسجيل في غضون 30 يوما. لا تقلق، إنه سريع و مجاني لكن يجب القيام بذلك.

اضغط على الرابط المُعطى لك، ستدخل موقع Microsoft. سجّل دخولك باستخدام Windows Live ID (المكافئ لحساب Hotmail أو MSN) أو قم بإنشاء واحد إذا لم يكن لديك، ثم أجب بعد ذلك على الأسئلة.

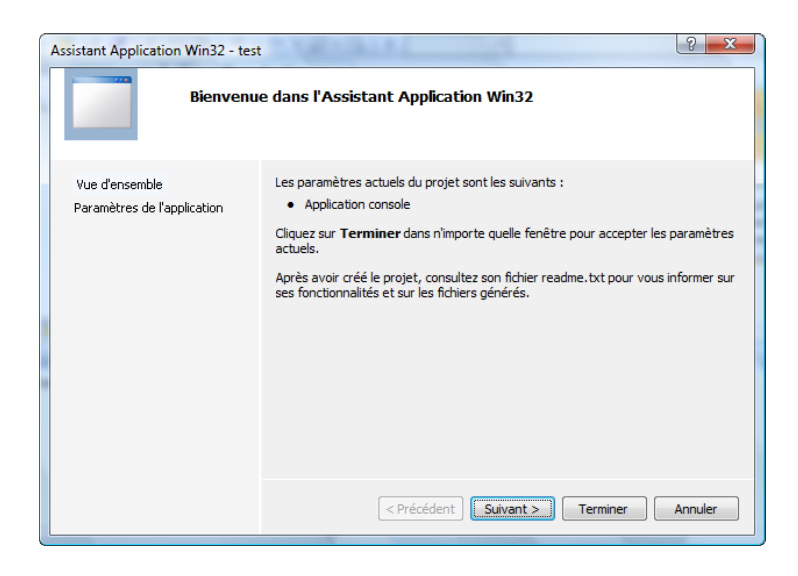
سيتم إعطاؤك في النهاية مفتاح تفعيل. انسخ هذا المفتاح في القائمة ? ثم "تسجيل المنتج".

إنشاء مشروع جديد

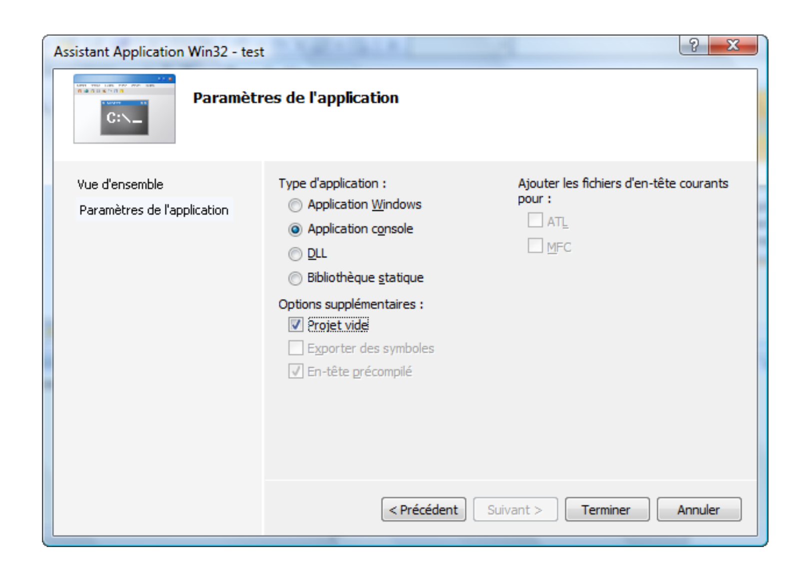
لإنشاء مشروع جديد، إذهب إلى قائمة "ملف" (`File`) ثمّ "جديد" (`New`) ثمّ "مشروع" (`Project`). اختر `Win32` في العمود الأيسر ثمّ `Win32 Console Application`. ثمّ أدْخِل اسم مشروعك.



وافق، ستظهر لك نافذة جديدة.



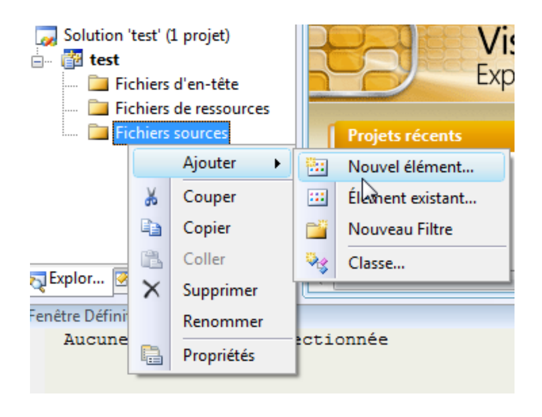
هذه النافذة لا تحوي أيّ شيء مهمّ، تابع فقط.



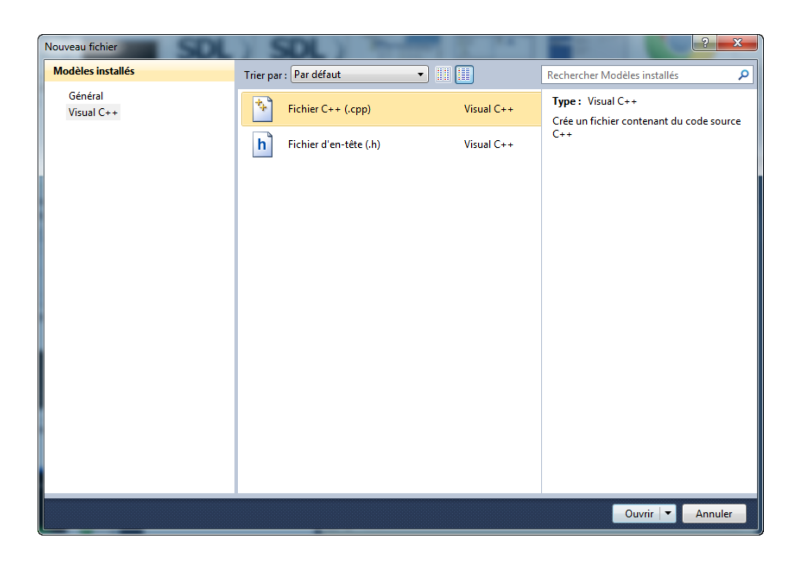
اختر `Console application` و تأكّد من إنشاء مشروع فارغ عن طريق تحديد `Empty project`، ثم اضغط على "إنهاء" (`Finish`).

إضافة ملف مصدري جديد

مشروعك فارغ لحدّ الآن. لإضافة ملف مصدري، اضغط باليمين على "الملفات المصدرية" (`Source files`) الموجود على اليسار، ثمّ اختر "إضافة" (`Add`) ثمّ "عنصر جديد" (`New element`).



اختر `Visual C++` على اليسار ثمّ `C++ File` (أعلمُ أنّنا لا ندرس C++ و لكن ليس لهذا أهميّة هنا). أدْخِل اسم الملف : `main.c`

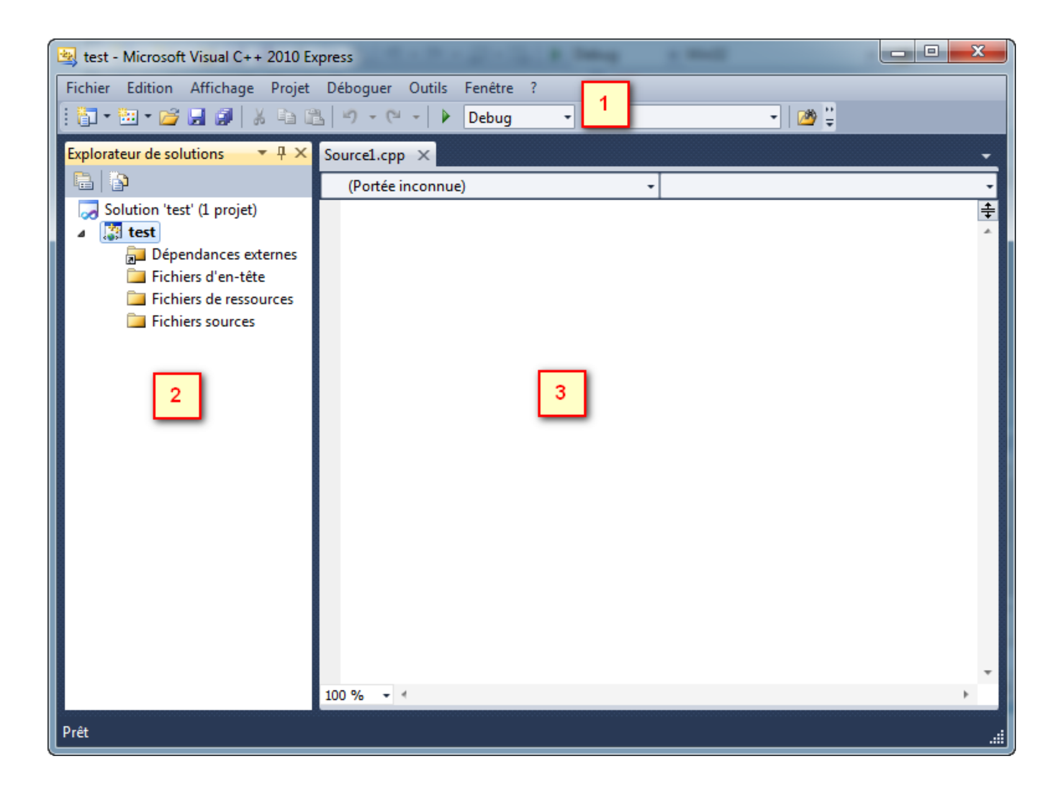


ثم اضغط على "إضافة" (`Add`). سيتم إنشاء ملفّ فارغ. أنصحك بحفظه بسرعة باسم `main.c`.

انتهى، يمكنك الآن أن تبدأ في كتابة الشفرة.

النافذة الرئيسيّة

لنرى ما هي أهمّ أقسام النافذة الرئيسيّة في Visual C++ Express.



هذه النافذة تشبه مثيلتها في Code::Blocks. و لكن رغم ذلك سوف نعيد رؤية معنى كلّ جزء.

1. شريط الأدوات : فيه أزرار اعتيادية. لكن كما ترى لا يوجد أيّ زرّ للترجمة. يمكنك إضافته عن طريق النقر باليمين على هذا الشريط و اختيار "تنقيح" (`Debug`) و "توليد" (`Generate`) من القائمة.

   كلّ هذه الأزرار لديها ما يكافئها في القوائم `Debug` و `Generate`. استخدام `Generate` ينشئ الملف التنفيذي (أي أنها تعني الترجمة). إذا استخدمت `Debug / Execute` فسوف يقترح عليك الترجمة قبل التشغيل. إختصارات لوحة المفاتيح : `F7` لتوليد المشروع و `F5` لتشغيله.

2. هذه المساحة جدّ مهمّة، إذ أنها تحتوي على الملفات الخاصة بمشروعك. أنقر على "مستكشف الحلول" (`Solution explorer`) في الأسفل إن لم يكن فُعِل من قبل. سوف ترى أنّه قد تمّ إنشاء مجلّدات لفصل أنواع الملفّات المختلفة (مصدريّة، رأسيّة و موارد). سنتعرف لاحقا على مختلف أنواع الملفات التي تكوّن المشروع.

3. المساحة الرئيسية : التي نعدّل فيها الملفّات المصدريّة.

أكملنا جولتنا في ++Visual C. يمكنك إلقاء نظرة على الخيارات إن أردت لكن لا تأخذ من وقتك ثلاث ساعات هناك ! لأنه يوجد كثير منها.

4.2 Xcode (Mac OS X فقط)

هناك الكثير من البيئات التطويرية المتوافقة مع Mac على غرار Code::Blocks طبعا.
سأقدم لك البيئة الأكثر شهرة في الماك و هي Xcode.

Xcode، أين أنت ؟

أغلب مستخدمي Mac OS X ليسوا مبرمجين. لقد فهمت Apple هذا، لذلك لم تثبّته افتراضيا مع النظام.
لحسن الحظ، لكلّ من يريد أن يبرمج، كلّ شيء جاهز. Xcode متوفّر على MacAppStore. ابدأ بأخذه من هناك.

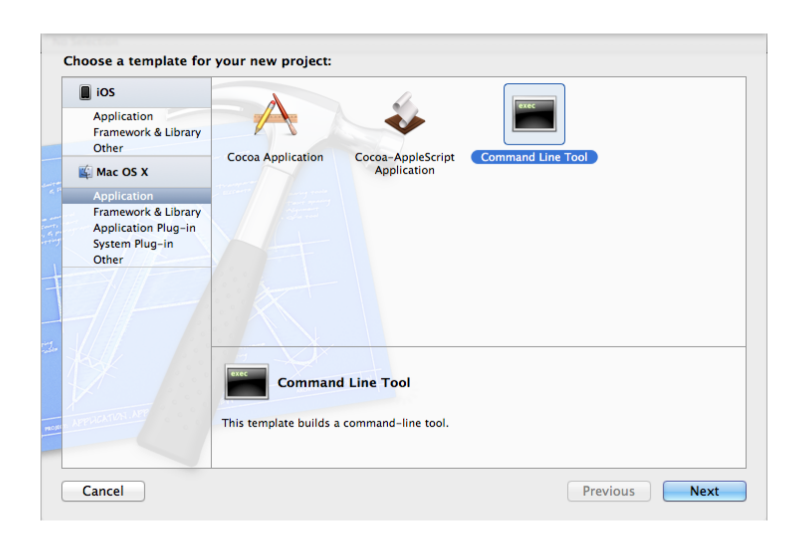
أنصحك أيضا بإلقاء نظرة على الموقع الخاص بالمطوّرين لـApple.

https://developer.apple.com/

سوف تجد هناك كمّا هائلا من من المعلومات المهمّة للتطوير على Mac. يمكنك منه تحميل العديد من البرامج للتطوير.
لا تتردّد في التسجيل في ADC (Apple Development Connection)، إنه مجاني و يساعدك على تتبع كل ما هو جديد.

تشغيل Xcode

أوّل شيء يمكننا فعله هو إنشاء مشروع جديد، فلنبدأ بهذا. إذهب إلى "ملف " (`File`) ثمّ "مشروع جديد" (`New Project`). ستفتح لك نافذة اختيار المشروع.

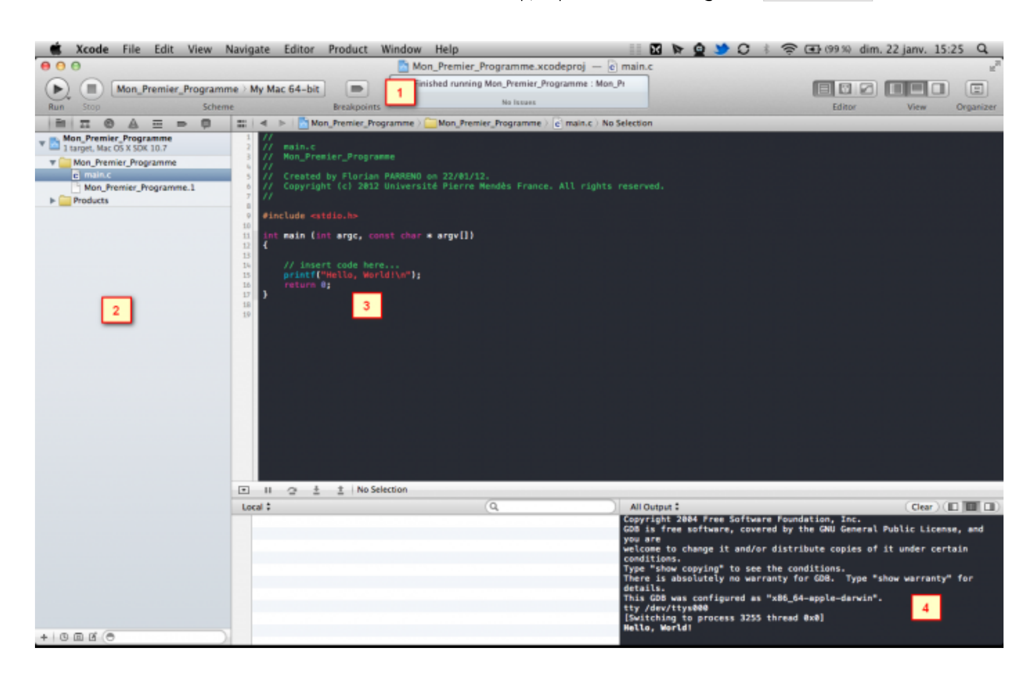


اختر `Application` من اليسار ثمّ `Command Line Tool`. اضغط بعدها على `Next`. سوف يُطلب منكم بعدها حفظ مشروعك (كلّ مشروع يجب أن يحفظ منذ البداية) و اسمه. ضعه في المجلّد الّذي تريد.

بمجرّد إنشائه، سيتمّ عرض مشروعك على شكل مجلّد يحتوي على العديد من الملفّات في الـ`Finder`. الملف الّذي يملك الامتداد `.xcodeproj` يوافق ملف المشروع. إنه الملف الذي عليك اختياره في المرة القادمة لفتح مشروعك.

نافذة التطوير

في Xcode، عندما تختار `main.c` تظهر لك نافذة شبيهة بهذه :



الواجهة مقسمة إلى أربعة أقسام، مرقّمة هنا من 1 إلى 4 :

1. الجزء الأوّل هو شريط الأزرار في الأعلى. أهمّ زرّ فيه هو "تشغيل" (`Run`) وظيفته تشغيل البرنامج.

2. الجزء اليسار مخصص للتمثيل الشُجيري لمشروعك الخاص. بعض الأقسام تحتوي على الأخطاء، التحذيرات، إلخ. يقوم `Xcode` تلقائيّا بنقلك إلى القسم المهم. و هو الذي يحمل اسم المشروع.

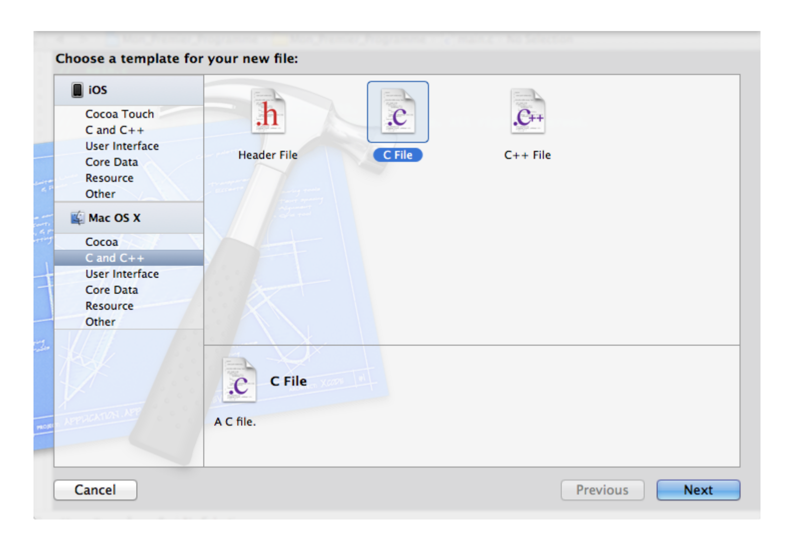
3. الجزء الثالث تتغيّر وظيفته حسب ما قمت بتحديده في الجزء الأيسر. و هنا يعرض محتوى الملف `main.c`.

4. أخيرا، الجزء الرابع يُظهر نتائج تشغيل البرنامج على الشاشة عندما تقوم بتشغيل البرنامج.

إضافة ملفّ جديد

في البداية، لن تملك سوى ملف مصدري واحد و هو `main.c`، و لكن لاحقا عندما نتقدم في الدروس سأطلب منك إنشاء ملفات مصدريّة بنفسك، عندما تصبح برامجنا أكبر.

لإنشاء ملفّ جديد، اذهب إلى قائمة `File` ثم `New File`. سيطلب منك إدخال نوع الملف الذي تريد إنشاءه. توجّه إلى قائمة `Mac OS X` و اختر `C and C++` ثم `C File`. لاحظ هذه الصورة.



يجب عليك إعطاء اسم للملّف الجديد. امتداده يجب أن يبقى `.c`. أحيانا -كما سنرى لاحقا- يجب عليك أيضا إنشاء ملفّات بامتداد `.h`. الخانة `Also create file.h` مخصّصة لهذا الغرض. حاليّا هذا الخيار لا يهمّنا.

انقر بعدها على `Finish`. انتهى ! أصبح في مشروعك ملفٌ آخر غير الملف `main.c`، هنيئا لك فقد أصبحت الآن جاهزاً للبرمجة على الـ `Mac`.

ملخّص

- المبرمجون يحتاجون إلى ثلاثة أدوات : محرّر نصوص، مترجم و منقّح.
- من الممكن تثبيت هذه الأدوات منفصلة، لكنّه من المعتاد اليوم الحصول على حزمةٍ ثلاثة-في-واحد نسميها بيئة التطوير المتكاملة.
- Code::Blocks، Visual C++ و Xcode تعدّ من بين بيئات التطوير الأكثر شهرة.

الفصل 3

برنامجك الأوّل

لقد قمنا بحضير كلّ شيء إلى حد الآن ويمكننا أن نبدأ قليلا من البرمجة. مع نهاية هذا الفصل ستكون قد نجحت في إنشاء أوّل برنامج لك.

لكي أصدقك القول، سيظهر البرنامج بالأبيض والأسود ولن يقوم بشيء سوى إلقاء التحيّة. يبدو عديم الفائدة، لكنّه برنامجك الأوّل وأؤكّد لك أنّك ستكون فخورا به.

1.3 كونسول أو نافذة ؟

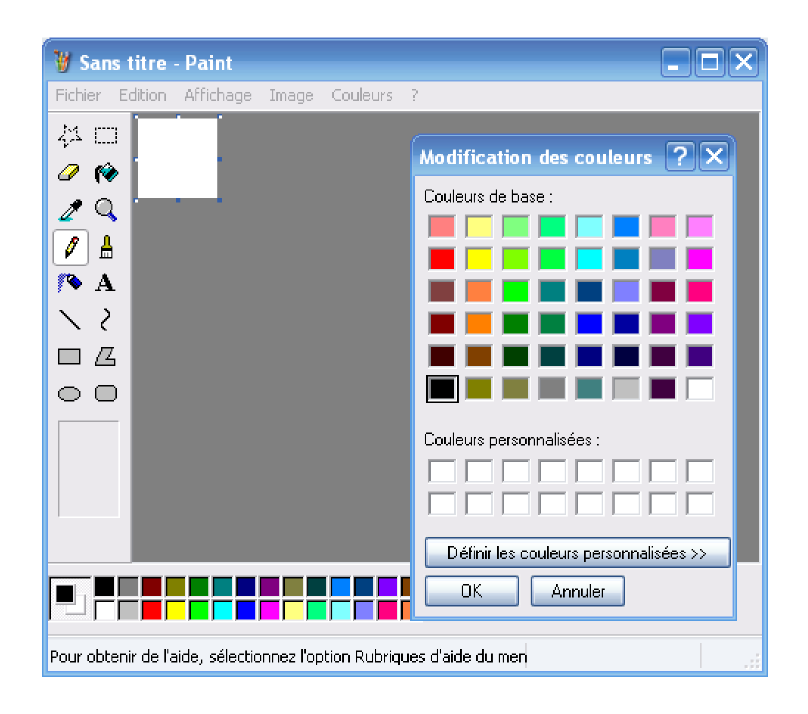
لقد تحدثنا سابقا عن فكرة برامج الكونسول وبرامج النوافذ في الفصل السابق. البيئة التطويرية تطلب منا تحديد أي نوع من البرامج نريد أن ننشئها. ولقد قلنا إننا سننشئ برامج من نوع كونسول.

يوجد نوعان من البرامج، لا أكثر :

- ،برامج بنوافذ
- برامج تعمل في الكونسول.

البرامج الّتي تملك نوافذ

هي البرامج التي نعرفها جميعا. هذا مثال على برنامج من نوع نافذة، مثل الرسام.



أعتقد أنّك تحب إنشاء برامج كهذه، لكنّ هذا ليس في مقدورك حاليا. في الواقع، إنشاء برامج بنوافذ هو أمر ممكن بلغة C، لكنّ بالنسبة لمبتدئ، هذا أمر معقّد جدّا. كبداية، يستحسن إنشاء برامج الكونسول.

> لكن ماذا يعني برنامج Console ؟

البرامج الّتي تعمل في الكونسول

برامج الكونسول هي أول ما ظهر من برامج. في ذلك الوقت، شاشات الحواسيب لم تكن سوى بالأبيض والأسود، ولم تكن فعّالة لكي تتمكّن من رسم النوافذ كما هو الحال مع حواسيبنا حاليّا.

مرّ الزمن بسرعة وزادت شعبية الويندوز نظراً لبساطته إلى أن نسي كثير من الناس ما هي الكونسول.

لديّ خبر جيّد لك ! **الكونسول لم تمت بعد** ! في الواقع، GNU/Linux قد أعاد الكونسول إلى الحياة. هذه صورة لكونسول على GNU/Linux.

```
2.2.5_appli.html      3.2.7.css         3.6.8.html
2.2.5.css             3.2.8.css         3.6.9.html
2.2.6_appli.html      3.2.9_appli.html  ancres.html
2.2.6.css             3.2.9.css         base.php
2.3.10_appli.html     3.3.10.html       cible_formulaire.php
2.3.10.css            3.3.11.css        cible.html
2.3.11_appli.html     3.3.12.css        design1.css
2.3.11.css            3.3.13_appli.html erreur_paragraphe.html
2.3.12.css            3.3.13.css        essai2.css
2.3.13.html           3.3.14_appli.html essai.css
2.3.14.css            3.3.14.css        images
2.3.15.html           3.3.15.css        tests_design.html
2.3.16.css            3.3.1.css         traitement.php
2.3.17.css            3.3.2.html
2.3.18.css            3.3.3.css
[root@ns1 exemples]# cd ..
[root@ns1 xhtml-css]# ls
anims                 css.php           images        pseudoformats.php
annexes               design.php        images.php    qcm.php
autres                exemples          index.php     tableaux.php
boites_partie1.php    formatage_partie1.php intro.php  texte.php
boites_partie2.php    formatage_partie2.php liens.php  xhtml.php
conclusion.php        formulaires.php   listes.php
[root@ns1 xhtml-css]#
```

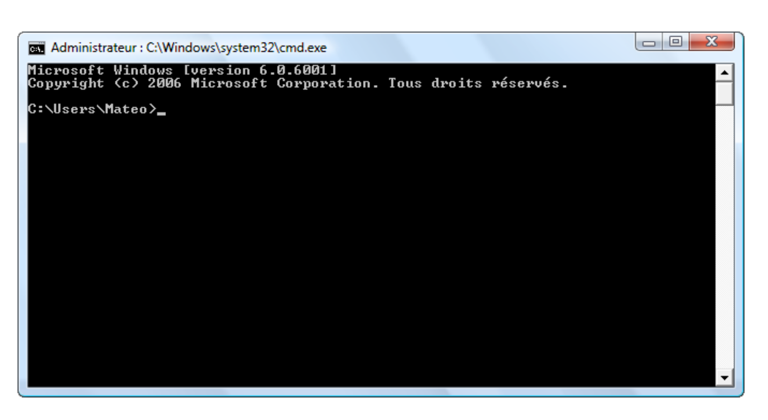
مرعب ! صحيح ؟ لكن على الأقل عرفت ما هي الكونسول، وهذه بعض الملاحظات :

- اليوم، يمكننا عرض الألوان في الكونسول. ليس كلّ شيء بالأبيض والأسود كما تخيّل.
- الكونسول هو الأسهل من ناحية البرمجة بالنسبة للمبتدئين.
- أداة عالية الإمكانيّات إذا عرفنا كيف نستخدمه.

كما قلت لك، إنشاء برامج كونسول أمر سهل جدّا وملائم للمبتدئين (وهذا عكس برامج النوافذ). ليكن في علمك أيضا أنّ الكونسول قد تطوّرت وبإمكانها عرض الألوان، ولا شيء يمنعك من إضافة صورة خلفيّة لها.

> ؟
> وفي الويندوز ألا توجد Console ؟

بلى، لكنّها مخفيّة لو صح القول. يمكنك فتحها بالذهاب إلى "إبدأ" (`Start`) ثمّ "ملحقات" (`Accessories`) ثمّ "موجه الأوامر" (`Command prompt`) أو بالذهاب إلى "إبدأ" ثمّ "تشغيل" (`Run`) واكتب فيها `cmd` واضغط على "موافق".



إذا كنت تستخدم نظام ويندوز، فاعلم بأن أولى برامجك ستكون في نوافذ شبيهة بهذه. أنا لم أختر البداية هكذا لجعلك تشعر بالملل، بل لتعليمك الأساسيّات اللازمة لكي تتمكّن لاحقا من إنشاء النوافذ.

إذن فلتكن متيقّناً، بمجرّد أن تصل إلى المستوى اللازم لإنشاء النوافذ، سوف أعلّمك كيف تفعل ذلك.

2.3 الحدّ الأدنى من الشفرة المصدرية

من أجل أي برنامج، يجب كتابة قدر معيّن من الشفرة المصدرية. هذه الشفرة لا تقوم بشيء خاصّ لكنّها ضروريّة. هذه الشفرة التي سنكتشفها الآن ستكون أساس أغلب برامجك الّتي ستكتبها بلغة C.

أطلب من البيئة التطويرية الخاصة بك تزويدك بالحد الأدنى من الشفرة المصدرية

لقد لاحظت أن طريقة إنشاء مشروع جديد تختلف من بيئة تطويرية إلى أخرى. إليك تذكيراً بسيطا : في برنامج Code::Blocks (الذي سنستخدمه في هذا الكتاب)، عليك التوجه نحو `File` ثمّ `New` ثمّ `Project` ثم تختار `Console Application` وبعدها اللغة C. سيولّد لك الحد الأدنى من الشفرة المصدرية C التي تحتاجها. ها هي :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Hello world!\n");
    return 0;
}

```

م

لاحظ أنّه يوجد سطر فارغ في نهاية الشفرة. يفترض أن ينتهي كل ملف مكتوب بلغة C هكذا. إن لم تفعل ذلك، فهذه ليست بمشكلة، لكن توقّع أن يعرض لك المترجم تحذيراً (Warning).

علماً أنّ السطر :

```
int main()
```

... بإمكانه أن يُكتب كالتالي :

```
int main(int argc, char *argv[])
```

كلتا العبارتين تحملان نفس المعنى لكن الثانية، الأكثر تعقيدا، هي الأكثر شيوعا، لذلك فإنّنا سنستخدمها في الفصول القادمة.
إستخدامنا للشكل الأوّل أو الثاني لا يغيّر شيئا بالنسبة لنا. لذلك لا داعي لإضاعة الوقت هنا، خصوصاً أنّك لا تملك المستوى اللازم لفهم ما تعنيه.

إذا كنت تستخدم بيئة تطويرية أخرى فقم بنسخ هذه الشفرة المصدرية وألصقها في الملف `main.c` ليكون لديكم نفس الشفرة.

أخيرا، قم بحفظ عملك في المشروع. أعلم أننا لم نقم بشيء حتّى الآن لكن من الجيّد التعوّد على الحفظ في كلّ مرّة.

تحليل أسطر الشفرة المصدرية السابقة

قد تبدو لك الشفرة المصدرية السابقة أنّها كاللغة الصينيّة، أنا أتخيّل ذلك ! في الواقع هي تسمح بإنشاء برنامج كونسول يعرض نصّا على الشاشة. يجب تعلّم كيفيّة قراءة كلّ هذا.

فلنبدأ بأوّل سطرين :

```
#include <stdio.h>
#include <stdlib.h>
```

هذان السطران يبدآن بعلامة `#`. وهي أسطر خاصّة تُعرف باسم **توجيهات المعالج القبلي** (Preprocessor directives). اسم معقّد، أليس كذلك ؟ هذه الأسطر تتمّ قراءتها من طرف البرنامج المسمّى بالمعالج القبلي، وهو برنامج يتمّ تشغيله في بداية الترجمة.

ما رأيناه سابقا كان مخطّطا بسيطا لعمليّة الترجمة. لكنّ في الواقع، هناك الكثير من المراحل التي تحدث في هذه العمليّة. سنقوم بتفصيل هذا لاحقا. حاليّا عليك فقط تذكّر وضع هذين السطرين أعلى كلّ ملفّاتك.

> ؟
>
> حسنا لكن ماذا يعنيه هذان السطران ؟ أريد أن أعرف !

كلمة `include` بالإنجليزيّة تعني "تضمين". هذان السطران يقومان بتضمين ملفّات في المشروع، أي إضافة هذه الملفّات من أجل عمليّة الترجمة. هناك سطران وبالتالي هناك ملفان يتمّ تضمينهما في المشروع وهما بالترتيب : `stdio.h` و `stdlib.h`. هذان الملفّان موجودان بالفعل على حاسوبك وهما ملفّان مصدريّان جاهزان، سوف تعرف مستقبلا أنّنا نسميها **مكتبات** (Libraries). هذه الملفّات تحتوي الشفرة المصدرية اللازمة لعرض نصّ على الشاشة.

بدون هذين الملفّين، كتابة نصّ على الشاشة سيكون أمرا مستحيلاً. فالحاسوب لا يعرف فعل أي شيء مبدئيا.

باختصار، السطران الأول والثاني يقومان بتضمين المكتبات التي ستساعدنا في إظهار نصّ على الشاشة بكلّ سهولة.

نمر للتالي، باقي الأسطر :

```c
int main()
{
    printf("Hello world!\n");
    return 0;
}
```

ما تراه هنا هو ما نسميه **بالتابع** أو **الدالّة** (Function). البرنامج في لغة C يتكوّن من مجموعة دوال. حاليّا برنامجنا لا يحوي سوى دالّة واحدة.

الدالّة تمكّننا من تجميع مجموعة من الأوامر. الغرض من تجميع الأوامر هو جعلها تقوم بوظيفة ما. مثلا يمكننا إنشاء دالّة باسم `open_file` وجعلها تحتوي التعليمات التي تشرح للحاسوب كيفيّة فتح ملف.

دون الدخول في تفاصيل إنشاء الدالّة (الوقت مبكّر، سوف نتحدّث عن الدوال في وقت لاحق) لنحلّل رغم ذلك أجزائه الكبيرة. السطر الأوّل يحتوي اسم الدالّة، إنّه الكلمة الثانية.
أجل، اسم دالّتنا هو `main` والذي يعني "الرئيسية" . وتشغيل البرنامج دائما يبدأ من الدالة `main`.

للدالّة بداية ونهاية، وهي محدودة بالحاضنتين `{` و `}`. محتوى الدالّة موجود بين هاتين الحاضنتين. إن كنت قد تابعت جيداً فقد عرفت أنّ الدالّة مشكّلة من سطرين :

```c
printf("Hello world!\n");
return 0;
```

هاته الأسطر في الداخل نسميها **التعليمات** (Instructions) (هذه إحدى المصطلحات الّتي يجب عليك حفظها). كلّ تعليمة تمثّل أمراً بالنسبة للحاسوب. فكلّ واحدة منها تطلب منه فعل شيء محدّد.

كما قلت لك، بتجميع ذكيّ للتعليمات في الدالّة يمكننا إنشاء أجزاء برنامج جاهزة للاستخدام. باستخدام التعليمات المناسبة يمكننا إنشاء دالّة `open_file` كما شرحت لك قبل قليل، و أيضا دالّة `move_character` في لعبة فيديو، على سبيل المثال.

البرنامج في الواقع ما هو إلّا تتابع لتعليمات : إفعل هذا و إفعل ذاك. أنت تعطي أوامر للحاسوب و هو يقوم بتنفيذها.

> X
>
> هامّ جدّا : لا بدّ أن تنتهي كلّ تعليمة بفاصلة منقوطة " `;` ". بهذا يمكن التفريق بين ما إذا كانت هذه تعليمة أم لا. إذا نسيت وضع فاصلة منقوطة نهاية تعليمة ما، فلن تتمّ ترجمة برنامجك.

السطر الأول : `printf("Hello world!n");` يطلب إظهار الرسالة "Hello world!" على الشاشة. عندما يصل برنامجك إلى هذا السطر، فسوف يقوم بعرض هذه الرسالة ثمّ المرور إلى التعليمة التالية.

التعليمة التالية هي `return 0;` و هي تخبرنا أنّ الدالّة `main` قد انتهت و تطلب منه إعادة 0.

> ؟
>
> لماذا يقوم برنامجي بإعادة العدد 0 ؟

في الواقع، كلّ برنامج عندما ينتهي يُرجع قيمة معينة. على سبيل المثال، ليقول أنّ كلّ شيء سار على ما يرام. عمليّا، 0 يعني أنّ كلّ شيء سار على ما يرام، و كلّ قيمة أخرى تدلّ على حدوث خطأ. في أغلب الأحيان هذه القيمة لا تُستخدم ، لكن يجب رغم ذلك استعمالها.
كان يمكن أن يعمل برنامجك بدون `return 0` ، لكن يمكننا القول أن وضعها يعتبر أمراً أكثر نظافة و أكثر جدّية.

إلى هنا نكون قد فصّلنا قليلا في عمل هذه الشفرة المصدرية.

طبعا، نحن لم ندرس كلّ شيء بعمق، و قد تكون لديك بعض الأسئلة عالقة في ذهنك. كن على يقين بأنك ستجد لها أجوبة شيئا فشيئا مع تقدّمنا في الكتاب. لا يمكنني أن أطلعك على كلّ شيء من البداية، لأنّ هناك كثيراً من الأشياء لاستيعابها.

إليك ما يلي : بما أنني في حال جيّدة، سأقوم بوضع مخطّط يضمّ المصطلحات الّتي تعلّمناها في هذا الفصل.

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Hello world!\n");
    return 0;
}
```

توجيهات المعالج القبلي

تعليمات (أوامر)

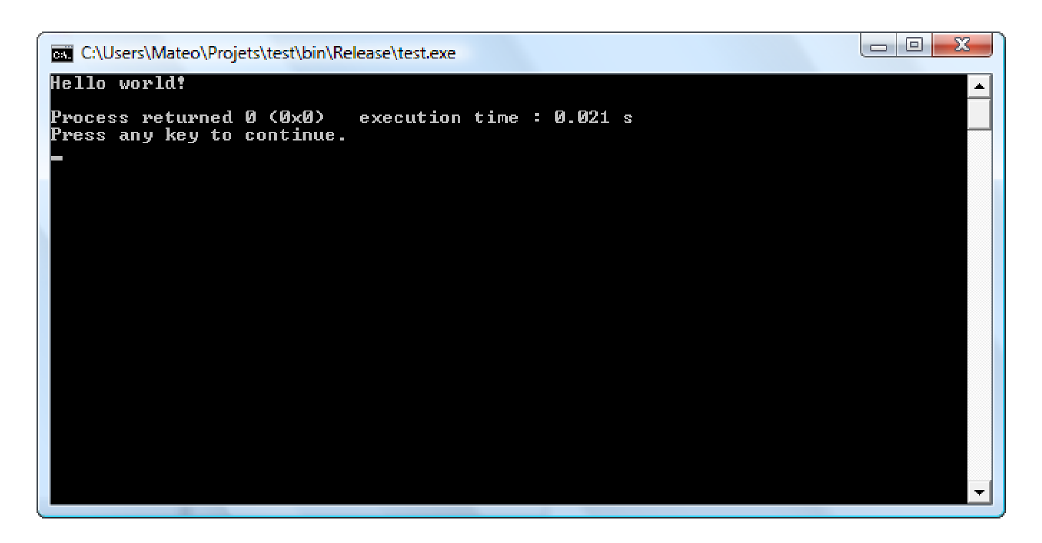
دالّة

لنجرّب برنامجنا

كلّ ما سنقوم به الآن هو ترجمة المشروع ثمّ تشغيله (اضغط على `Build & Run` إذا كنت على Code::Blocks).
سيطلب منك حفظ مشروعك إذا لم تقم بذلك من قبل.

> X
> إن لم تنجح الترجمة و ظهر لك خطأ مثل :
> `"My-program - Release" uses an invalid compiler. Skipping...`
> `Nothing to be done...` فهذا يعني أنّك نزلت نسخة Code::Blocks دون `mingw` (المترجم)، عد و نزّل النسخة التي تحتوي على `mingw`.

بعد بُرهة، يظهر برنامجك كما في الصورة :



البرنامج يُظهر "Hello world!" (في السطر الأوّل).
الأسطر الّتي أسفله تمّ توليدها من طرف Code::Blocks وتدلّ على أنّ البرنامج قد تمّ تشغيله بنجاح كما أنها تعطي الوقت الذي استغرقه البرنامج في التشغيل.

سيطلب منك الضغط على إحدى المفاتيح لإغلاق النافذة. أعلم أن الأمر لم يكن ممتعا جدّا. لكنه برنامجك الأوّل، وهذه لحظة ستتذكرها طيلة حياتك ! ألا تعتقد ذلك ؟

3.3 كتابة رسالة على الشاشة

من الآن سنقوم بإدخال التعديلات على الشفرة المصدرية السابقة. مهمّتك، إن قبلتها : عرض رسالة "Bonjour" على الشاشة.

> ؟
> كيف يمكنني اختيار النص الّذي سيظهر على الشاشة ؟

الأمر بسيط جدا، إذا بدأت من الشفرة التي رأيناها سابقاً، فسيكون عليك استبدال "Hello world!" بـ"Bonjour" في السطر الذي يستدعي `printf`.

كما قلت من قبل، `printf` هي **تعليمة** وهي تعطي أمراً للحاسوب : "قم بعرض هذه الرسالة على الشاشة".
يجب أن تعرف أيضا أن `printf` هي دالّة كُتِبَت من قبل من طرف مبرمجين قبلك.

> ؟ أين توجد هذه الدالّة ؟ أنا لا أرى سوى الدالّة `main` !

هل تذكر هذين السطرين ؟

```
#include <stdio.h>
#include <stdlib.h>
```

قلت لك من قبل أنهما يمكّان البرنامج من إضافة مكتبات. المكتبات في الحقيقة هي ملفّات تحوي أطنانا من الدوال جاهزة للإستخدام. هذه الملفات (`stdio.h` و `stdlib.h`) تحوي أغلب الدوال الأساسية التي قد نحتاجها في برنامج ما. `stdio.h` بحد ذاته يحوي دوال تمكّن من عرض أشياء على الشاشة (مثل `printf`) و أيضا الطلب من المستخدم إدخال شيء ما (هذه دوال سنتعرّف عليها لاحقا).

لنقل مرحبا للسيّد

في دالّتنا `main` نستدعي الدالّة `printf`. أي أن لدينا دالّة تستدعي أخرى (هنا `main` تستدعي `printf`). سترى أن هذا ما يحدث دائما في لغة C : دالّة تحتوي تعليمات تستدعي دوال أخرى، وهكذا.

إذن، لاستدعاء دالّة يكفي كتابة اسمها متبوعا بقوسين، ثم فاصلة منقوطة.

```
printf();
```

هذا جيد، لكنه غير كاف. يجب أن نُعلِم البرنامج بما يجب أن يكتبه في الشاشة. لفعل هذا يجب أن نعطي `printf` النص المطلوب عرضه. لفعل هذا نقوم بوضع النص داخل علامات الإقتباس المزدوجة بين القوسين. في حالتنا هذه سنكتب تماما :

```
printf("Bonjour");
```

آمل ألا تكون قد نسيت رمز الفاصلة المنقوطة في النهاية، وأذكّرك أنّها مهمّة جدا لأنّها تدلّ على نهاية التعليمة.
هذه هي الشفرة المصدرية التي يجب أن تحصل عليها :

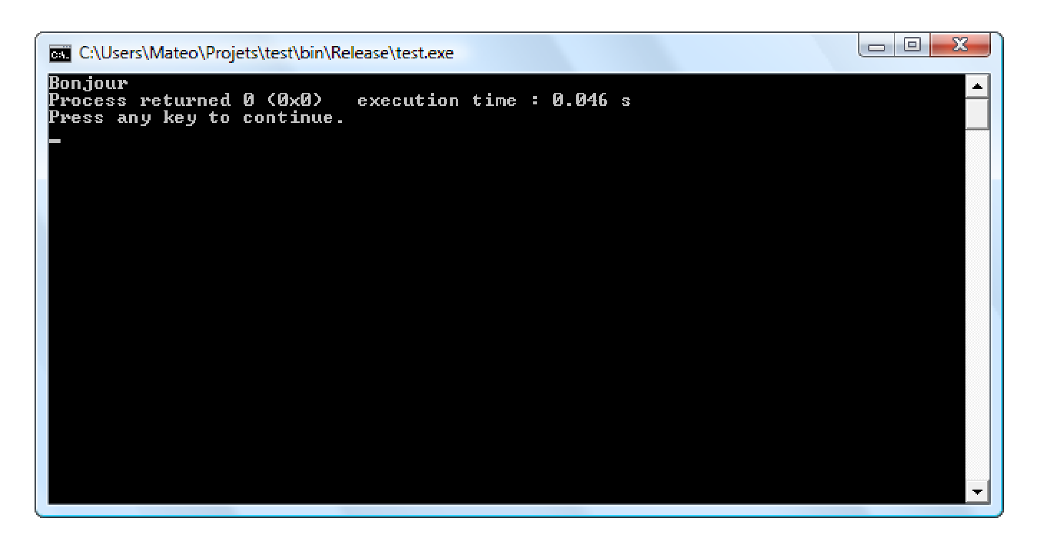
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Bonjour");
    return 0;
}
```

لدينا إذن تعليمتان تطلبان من الحاسوب القيام بهذين الأمرين بهذا الترتيب :

1. عرض "Bonjour" على الشاشة.
2. نهاية الدالّة `main` ، إعادة 0. البرنامج يتوقّف.

هذا ما يظهر على شاشتك :



كما ترى، السطر الذي يحتوي الرسالة يكون ملتصقاً قليلا بباقي النص، على خلاف ما رأيناه سابقا.
أحد الحلول الممكنة هو إضافة رمز للعودة إلى السطر بعد "Bonjour" (كما لو أنّنا ضغطنا على المفتاح `Enter`).

ولكن ضغط المفتاح `Enter` في الشفرة المصدرية لن يعمل كما نتوقع، لهذا يجب استخدام المحارف الخاصّة (Special characters).

المحارف الخاصّة

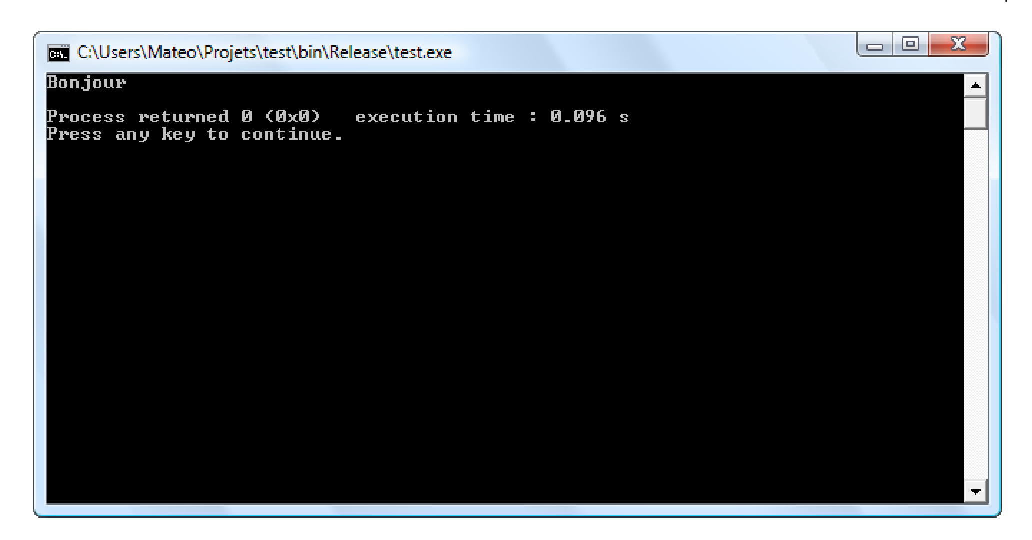
المحارف أو الرموز الخاصّة هي محارف تمكّن من تعريف عودة إلى السطر، جدولة، إلخ.
من السهل التعرّف عليها، فهي مكوّنة من محرفين. الأوّل هو الشَرْطَةُ المائلة الخلفية (\) (Backslash) والثاني يكون رقما أو حرفا. إليك محرفين خاصّين قد تحتاجهما كثيرا :

- `\n` : العودة إلى السطر.
- `\t` : الجدولة (فراغ كبير في نفس السطر).

في حالتنا هذه، يكفي أن نكتب `\n` لإنشاء العودة إلى السطر. إذن، إذا أردنا أن نضع عودة إلى السطر بعد Bonjour ، فيكفي أن نكتب :

```
printf("Bonjour\n");
```

وسيفهم حاسوبك أنّ عليه كتابة "Bonjour" ويعود إلى السطر.



يمكنك الكتابة بعد `\n` بدون أيّة مشكلة. كلّ ما تكتبه بعد `\n` سيوضع في السطر الجديد. يمكنك إذن التدرّب على كتابة : `printf("Good morning\nGood bye\n");`
و سيتمّ عرض "Good morning" على السطر الأوّل و "Good bye" على السطر الثاني.

متلازمة Gérard

مرحبا، اسمي Gérard و قد حاولت تعديل برنامجك ليقول "Bonjour Gérard"، و لكنّي ألاحظ أنّ حرف é لا يظهر بشكل جيّد ... مالّذي علي فعله ؟

أوّلا، مرحبا بك Gérard . هذا سؤال جيّد. لكن لديّ خبر سيّء لك. الكونسول الخاصة بـWindows لا تمكّن من عرض الحروف الّتي تحوي علامات النطق الصوتي مثل é ، خلافا لكونسول GNU/Linux التي تفعل. لديّ حلّان لهذه المشكلة :

- **استخدم GNU/Linux** . هذا حلّ جذريّ بعض الشيء. أحتاج إلى درس كامل لأعلّمك كيف تعمل على GNU/Linux . إذا لم يكن لديك المستوى، إنس هذا الخيار حاليّا.
- **لا تستخدم الحروف الّتي تحوي علامات النطق الصوتي.** للأسف إنّه الحل الذّي قد يكون عليك اختياره. الكونسول الخاصة بـWindows لها عيوبها. يجب عليك التعوّد على عدم كتابة مثل هذه الحروف. لكن مستقبلا قد تنشئ برامج بنوافذ ولن تعاني من هذا المشكل. لذلك أنصحك بالصبر على هذه المشكلة حاليّا، فبرامجك المستقبلية "الاحترافية" لن يكون فيها هذا المشكل.

لكيلا تنزعج، يمكنك الكتابة دون استخدام الحروف التي تملك علامات النطق الصوتي :

```c
printf("Bonjour Gerard\n");
```

نشكر صديقنا Gérard لتنبيهنا على هذه المشكلة !

4.3 التعليقات، مهمّة جدا !

قبل ختم هذا الفصل الأوّل "الحقيقي" في البرمجة، يجب أن أعرّفك على **التعليقات** (Comments) . أيّا كانت لغة البرمجة الّتي تستخدمها، ستكون لديك القدرة على إضافة التعليقات للشفرة المصدرية الخاصة بك.

ولكن ما الذي يعنيه "التعليق"؟
هذا يعني إمكانية وضع نصّ في وسط برنامجك لشرح دوره، مثلاً : ما الذي يفعله هذا السطر، إلخ. هذا بالفعل أمر ضروريّ، لأنّه حتّى لو كنت عبقرياً في البرمجة، ستكون بحاجة إلى وضع ملاحظات هنا وهناك. هذا يمكنك من :

- العثور على ما تحث عنه بسهولة في الشفرة المصدرية عندما تعود إليه بعد مدّة. من الطبيعيّ أن ننسى كيف تعمل البرامج الّتي كتبناها بعد مدّة. إن توقّفت عن البرمجة لأيّام ثمّ عدت فستكون بحاجة إلى التعليقات لإيجاد ما تريد في شفرة كبيرة جدّا.
- إذا أعطيت مشروعك لأحد غيرك (وهو لا يعرف شيئا عن الشفرة المصدرية الخاصة بك)، فالتعليقات تمكّنه من التآلف مع مشروعك بسرعة.
- وأخيرا، ستسمح لي بإضافة شروحات وملاحظات حول الشفرة المصدرية في هذه الدروس. وهذا سيفيدك في فهم ما الذي يعنيه كلّ سطر.

توجد طريقتان لإضافة تعليق. وهذا يعتمد على طول التعليق المراد إدراجه :

- إذا كان تعليقك **قصيرا** : فيمكن كتابته على سطر واحد، ولا يحتوي سوى كلمات قليلة. في هذه الحالة، عليك كتابة شرطتين مائلتين (`//`) متبوعين بتعليقك. على سبيل المثال :

```
// This is a comment.
```

بإمكانك إضافة تعليق وحده على السطر، أو على يمين تعليمة معينة. وهذا أمر مهمّ جدّا، لأنّ بهذه الطريقة يمكننا تحديد ما الذي يعنيه السطر الّذي كُتب بجانبه. مثال :

```
printf("Bonjour"); // This instruction displays 'Bonjour' on the screen
```

- إذا كان تعليقك **طويلا**: لديك الكثير لتقوله، تريد كتابة الكثير من الجمل على كثير من الأسطر. في هذه الحالة، يجب عليك كتابة شفرة تشير إلى "بداية التعليق" وأخرى تشير إلى "نهاية التعليق":
  - لبدء التعليق : أكتب شرطة مائلة متبوعة بنجمة (`/*`).
  - لإنهاء التعليق : أكتب نجمة متبوعة بشرطة مائلة (`*/`).

  يمكنك كتابة هذا على سبيل المثال :

```
/* This is
a comment
written on several lines */
```

فلنعد إلى الشفرة المصدرية التي تُظهر "Bonjour" على الشاشة ونضيف إليها بعض التعليقات للتدرّب :

```c
/*
Below, the directives of preprocessor.
These lines allow you to add files to your program,
files that we call libraries. Thanks to these libraries, we are ready to use
    functions for display.
for example, a message on screen.
*/

#include <stdio.h>
#include <stdlib.h>

/*
Following, you have the principal function of the program, called main.
All programs start with this function.
Here, all what does my function is displaying "Bonjour" on the screen.
*/

int main()
{
  printf("Bonjour"); // This instruction displays 'Bonjour' on the screen
  return 0;          // The program returns 0 then it stops.
}
```

هذا هو برنامجنا مع إضافة بعض التعليقات، نعم هو يبدو أكبر نوعا ما، لكنّه في الحقيقة مكافئ للبرنامج السابق. عند الترجمة، كلّ التعليقات يتمّ تجاهلها من طرف المترجم. هذه التعليقات لا تظهر في البرنامج النهائي، فهي تصلح فقط للمبرمجين.

عادة لا نقوم بوضع تعليق لكلّ سطر. لقد قلت وأكرر أنّه من المهم وضع التعليقات في الشفرة المصدرية، لكن يجب عليك معرفة القدر اللازم من التعليقات الواجب وضعه، وضع تعليق في كلّ سطر قد لا يفيد في شيء، بل يضيّع الوقت فقط. مثلا، أنت تعرف أن وظيفة `printf` هي عرض نصّ على الشاشة، فلا حاجة لوضع تعليق يشرح ذلك في كلّ مرّة.

من الأحسن التعليق عن عدد من الأسطر دفعة واحدة. هذا يفيد في ذكر وظيفة مجموعة من التعليمات المتتابعة. فيما بعد إن أراد المبرمج إضافة مزيد من التفاصيل في تعليماته، فسيكون بمستوى ذكاء يسمح له بفعل ذلك.

تذكر إذن: يجب أن تكون التعليقات لإرشاد المبرمج في شفرته المصدرية. حاول التعليق عن مجموعة من الأسطر دفعة واحدا بدل التعليق عن كلّ سطر على حدة.

وإليك هذه المقولة من IBM :

’إذا قرأت التعليقات الموجودة في برنامج و لم تفهم مبدأ عمله، قم برميه !

ملخّص

- البرامج يمكنها التفاعل مع المستخدم عن طريق الكونسول أو عن طريق النافذة.

- من السهل على المبرمج في برامجه الأولى استخدام **الكونسول**، رغم أنّ هذه قد تكون غير محبوبة لدى المبتدئ، فهذا لا يمنع من استخدام النوافذ في الجزء الثالث من هذا الكتاب.
- البرنامج يتكوّن من **تعليمات** تنتهي دائما بفاصلة منقوطة.
- الدالة `main` (التي تعني الرئيسيّة) هي الدالة الّتي يبدأ بها تنفيذ البرنامج. إنّها الدالة الوحيدة الإجبارية في البرنامج، لا يمكن لأي برنامج أن يُترجم بدونها.
- `printf` هي دالة تمكننا من عرض رسالة على الشاشة.
- `printf` موجودة في **مكتبة** تحتوي على كثير من الدوال الأخرى الجاهزة للاستخدام.

الفصل 4

عالم المتغيّرات

تعلّمت كيفية إظهار نصّ على الشاشة. جيد، لكنّ هذا ليس شيئا مهماً. هذا لأنك لا تعرف بعد ما يدعى بـ**المتغيّرات** (Variables) في البرمجة.

فائدة هذه المتغيرات هي تمكين الحاسوب من حفظ أعداد في الذاكرة. سنبدأ ببعض الشرح حول ذاكرة الحاسوب وكيفيّة عملها. قد يبدو هذا بسيطا جدّا للبعض، لكنّي أفترض أنّك لا تعرف شيئا عن ذاكرة الحاسوب.

1.4 أمر متعلق بالذاكرة

ما سأعلمك في هذا الفصل هو أمر له علاقة مباشرة بذاكرة حاسوبك.

كل إنسان حيّ له ذاكرة. الأمر عينه بالنسبة للحاسوب، لكن الحاسوب له أنواع عديدة من الذاكرة.

> ؟ لم يملك الحاسوب أنواع عديدة من الذاكرة، واحدة يمكنها أن تكفي، أليس الأمر كذلك ؟

كلّا : المشكلة أننا نحتاج ذاكرة سريعة (لاسترجاع المعلومات بسرعة) وفي نفس الوقت كبيرة (لحفظ بيانات كثيرة) قد تضحك إن أخبرتك أننا حتى اليوم لم نتمكن من صنع ذاكرة بهذه المواصفات. أو بالأحرى الذاكرة السريعة باهظة الثمن لذلك لا يتم إنتاج الكثير منها.

لذلك نجد في الحواسيب الحديثة ذاكرة سريعة جدا لكنها ليس ذات سعة كبيرة، وأخرى ذات سعة كبيرة جدّا لكنها غير سريعة.

الأنواع المختلفة من الذاكرة

كي أوضح لك الصورة أكثر، إليك أنواع الذاكرة الموجودة في الحاسوب، من الأسرع إلى الأبطأ :

1. السجلات (Registers) : ذاكرة سريعة جدّا، موجودة داخل المعالج.
2. ذاكرة التخبئة (Cache memory) : تمثل همزة وصل بين السجلات والذاكرة الحية.

3. ذاكرة الوصول العشوائي (Random access memory) : و هي الذاكرة التي نستخدمها كثيرا، وتدعى اختصارا RAM.

4. القرص الصلب (Hard disk) : والذي تعرفه بالطبع، نستعمله لحفظ الملفات.

كما قلت لك، لقد رتبتها من الأسرع (السجلات) إلى الأبطأ (القرص الصلب)، وإن كنت قد تابعت جيدا فقد فهمت أن الذاكرة الأصغر هي الأسرع والأبطأ هي الأكبر.
السجلات لا تسع إلا لحمل بضعة أعداد أما القرص الصلب فيمكنه تخزين ملفات ضخمة.

> عندما أقول ذاكرة بطيئة فهذا بالنسبة لحاسوبك، ففي نظر الحاسوب استغراق 8 ميلي ثانية للوصول إلى القرص الصلب يعتبر زمنا طويلا جدّا !

ما الذي يجب أن أتذكره من كل هذا ؟
أردت أن أخبرك أننا في الفصول القادمة سوف نستخدم ذاكرة الوصول العشوائي كثيرا. سنتعلم أيضا كيفية القراءة والكتابة في الملفات على القرص الصلب (ليس الآن، لا يزال الوقت مبكّرا على هذا). أمّا بخصوص السجلّات وذاكرة التخبئة فلن نتعامل معهما مطلقا، فالحاسوب هو من سيهتم بأمرهما.

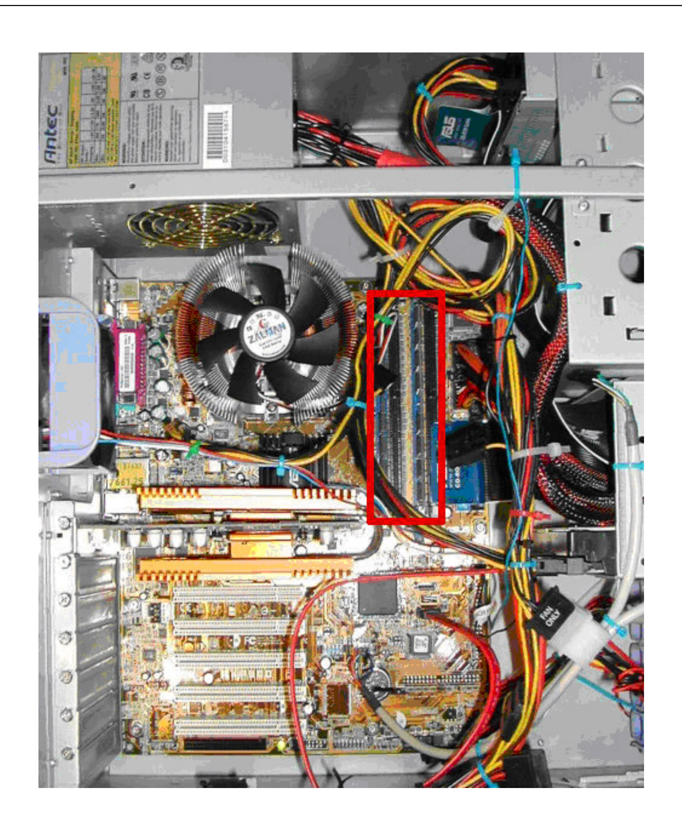
> في لغات البرمجة منخفضة المستوى، كلغة التجميع (Assembly language) نتعامل مباشرة مع السجلّات، لقد درستها، ويمكنني أن أقول لك أن القيام بعملية ضرب بسيطة يتطلب مجهودا ! لحسن الحظ ففي لغة C (وفي أغلب اللغات الأخرى) الأمر أسهل من ذلك بكثير.

يجب إضافة شيء مهمّ آخر : القرص الصلب هو الوحيد الذي يمكنه حفظ المعلومات بشكل دائم. **كل أنواع الذاكرات الأخرى مؤقتة، فبمجرد إطفاء الحاسوب تفقد كل محتواها !**

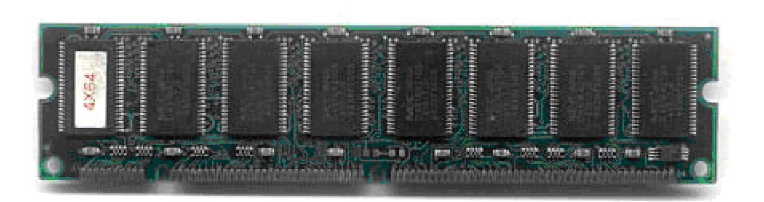
لحسن الحظ، عند إعادة تشغيل الحاسوب، يقوم القرص الصلب بتذكيرها بمحتواها.

صورة لذاكرة الوصول العشوائي

نظرا لأننا سنستعمل ذاكرة الوصول العشوائي خلال لحظات، فمن الأفضل أن أريها لك (مؤطرة بالأحمر):



لا أطلب منك معرفة كيفية عملها، لكن أردت فقط أن أريك مكانها داخل جهازك. وهذه صورة مقربة لإحدى أشرطتها :



و هي تدعى اختصارا **RAM**، لذلك لا تحتر إن سميتها هكذا لاحقا. بالنسبة للذاكرات الأخرى (السجلات والتخبئة) فهي صغيرة لدرجة أنه لا يمكن رؤيتها بالعين المجرّدة.

مخطط ذاكرة الوصول العشوائي

عرض المزيد من الصور لن يفيدك كثيرا، لكن يجب عليك فهم كيف تعمل من الداخل، لذلك سأقدم لك هذا المخطط البسيط الذي يمثل هندسة ذاكرة الوصول العشوائي :

العنوان	القيمة
0	145
1	3.8028322
2	0.827551
3	3901930
...	...
3 448 765 900 126 تقريبا	940.5118

كما ترى، يمكننا أن نميز عمودين :

- هناك **العناوين** : هي أعداد تسمح للحاسوب بتحديد موضع القيم في الـRAM . نبدأ بالعنوان 0 وننتهي بالعنوان 3,448,765,900,126 وبعض الأجزاء. لا أعلم بالضبط كم عدد العناوين الموجودة في الـRAM ، لكني أعرف أنها كثيرة جدا. إضافة إلى ذلك، هذا أمر يتعلق بكمية الذاكرة الموجودة في جهازك، فكلما زادت الذاكرة زادت معها العناوين وصار بإمكاننا تخزين معلومات أكثر.
- عند كل عنوان يمكننا تخزين **قيمة** (عدد). حاسوبك يقوم بتخزين هذه الأعداد في ذاكرة الوصول العشوائي لكي يتمكن من تذكرها. ولا يمكننا تخزين سوى عدد واحد عند كل عنوان.

لا يمكن للذاكرة الحية تخزين شيء سوى الأعداد.

؟

لكن كيف يمكننا تخزين الكلمات ؟

سؤال جيد. في الواقع حتى الحروف ليست سوى أعدادا في نظر الحاسوب ! الجملة هي مجرد تتابع لأعداد.
يوجد جدول يوافق بين الأعداد والحروف، جدول يقول مثلا بأن العدد 67 يوافق الحرف Y. لن أدخل في التفاصيل أكثر، ستكون لنا فرصة للرجوع إلى هذا لاحقا.

فلنعد إلى مخططنا، الأمور بسيطة جدا : إذا أراد الحاسوب تذكر العدد 5 (الذي قد يمثل عدد الأرواح المتبقية لشخصية في لعبة) فسوف يضعه في مكان ما في الذاكرة أين يتوفر مكان شاغر ويحفظ العنوان الموافق (مثلا 3,062,199,902). لاحقا، عندما يريد معرفة هذا العدد فسيذهب إلى خانة الذاكرة التي تحمل العنوان رقم 3,062,199,902 وسيجد القيمة 5.

هذه آلية عمل الذاكرة بشكل عام. قد يكون الأمر لا زال غامضا في ذهنك حاليا (ما فائدة تخزين عدد إن كان علينا تذكر عنوانه بدلا من ذلك ؟) لكن كل شيء سيتضح مع بقية الفصول، أنا أعدك !

2.4 التصريح عن متغير

صدّقني هذه المقدّمة القصيرة عن الذاكرة ستكون مهمّة أكثر مما تعتقد. الآن يمكننا العودة إلى البرمجة.

إذن، ما هو **المتغير** (Variable) ؟
إنه معلومة صغيرة نخزنها مؤقتا في الذاكرة الحية. ببساطة يمكننا القول إن المتغير هو قيمة يمكن أن تتغير أثناء اشتغال البرنامج. مثلا عددنا 5 الذي ذكرناه سابقا يمكن أن يتناقص بمرور الزمن. إذا وصل إلى العدد 0 فسنعرف أن اللاعب قد خسر.

في برامجنا سيكون هناك الكثير من المتغيرات. ستراها في كلّ مكان.

في لغة الـC، المتغير يتميز بشيئين :

- **قيمة** : هو العدد الذي يحويه، 5 مثلا.
- **اسم** : وهو الذي يمكننا من معرفة المتغيّر. في البرمجة لن يكون علينا تذكّر عناوين الذاكرة. بدلا من ذلك علينا فقط استخدام أسماء المتغيرات. المترجم هو من سيقوم بتحويل الأسماء إلى عناوين.

إعطاء اسم للمتغير

في لغة البرمجة C كل متغير يجب أن يملك اسما خاصا به. ومن أجل متغيرنا الذي يحوي عدد الأرواح المتبقية للاعب يمكننا أن نسميه "Number of lives" أو شيء من هذا القبيل.

للأسف توجد بعض الشروط، لا يمكنك تسمية المتغير كيفما شئت :

- لا يجب أن يحتوي الاسم سوى على الحروف الصغيرة والكبيرة والأرقام (`abcABC012`).
- يجب أن يبدأ الاسم بحرف.
- المسافات ممنوعة. بدلا من ذلك يمكننا استخدام الحرف المعروف باسم underscore (`_`). إنه الحرف الخاص الوحيد غير الحروف والأرقام الذي يمكن استعماله في اسم متغير.
- لا يمكنك استخدام حروف غير الحروف الإنجليزية.

وأخيرا يجب أن تعرف أن لغة C تفرّق بين الحروف الصغيرة والكبيرة. ولثقافتك، نقول إن C حساسة لحالة الحروف (Case sensitive). كمثال، الأسماء `width` أو `WIDTH` أو `WiDth` تعتبر أسماء متغيرات مختلفة، حتى لو كانت تعني لنا الأمر نفسه.

هذه أمثلة عن أسماء متغيرات صالحة : `numberOfLives`، `name`، `surname`، `phone_number`، `phoneNumber`.

لكل مبرمج طريقة خاصة في كتابة أسماء المتغيرات. خلال هذا الفصل سأريك طريقتي :

- أبدأ دائما بحرف صغير.

- إن كان في الاسم أكثر من كلمة أضع حرف كبيرا في بداية كلّ كلمة.

أطلب منك كتابة أسماء متغيراتك بنفس الطريقة التي أتبعها، هذا لكي نكون على تفاهم.

> X
> أيّا كان اختيارك، فعليك دائما إعطاء أسماء واضحة لمتغيراتك. كان بإمكاننا اختصار `numberOfLives` إلى `nol` مثلا. هذا أقصر في الكتابة، لكنه أقل وضوحا عندما تعيد قراءة الشفرة المصدرية. فأنصحك بإعطاء أسماء أطول لمتغيراتك إن كان ذلك يحسّن فهمها.

أنواع المتغيرات

حاسوبنا كما نعلم ليس سوى آلة كبيرة جدا للحساب. لا يجيد التعامل سوى مع الأعداد. لكن يوجد أنواع كثيرة من الأعداد :

- الأعداد الصحيحة الموجبة (الطبيعية) مثل :45، 398، 7650.
- الأعداد العشرية، أي التي تحوي فاصلة عشرية : 75.909، 1.7741، 9810.7.
- الأعداد الصحيحة السالبة : -87، -916.
- الأعداد العشرية السالبة : -76.9، -100.11.

حاسوبك المسكين بحاجة للمساعدة ! عندما تطلب منه تخزين عدد، يجب أن تذكر له نوعه. هذا ليس لأنّه لا يمكنه التعرف عليه تلقائيّا، ولكن للتنظيم ولعدم أخذ كميات كبيرة من الذاكرة بدون فائدة.

عندما تصرّح عن متغيّر فسيكون عليك تحديد نوعه. إليك أنواع المتغيرات الأساسية في لغة C :

النوع	الحد الأدنى	الحد الأقصى
`signed char`	-128	127
`int`	$-32,768$	$32,767$
`long`	-2147483648	2147483647
`float`	-1×10^{37}	$1 \times 10^{37} - 1$
`double`	-1×10^{37}	$1 \times 10^{37} - 1$

> !
> القيم المعروضة هنا تمثل الحد الأدنى المضمون من طرف اللغة. في الحقيقة قد تمكن من تخزين أعدادا أكبر من هذه. في كلّ الأحوال من المستحسن تذكّر هذه القيم عندما تختار نوع متغيراتك.

م لعلم أنّي لم أعرض جميع الأنواع هنا، بل الأساسية منها فقط.

الأنواع الثلاثة الأولى (`char`، `int`، `long`) تسمح بتخزين الأعداد الصحيحة (1، 2، 3، 4، ...).
النوعان الأخيران (`float`، `double`) يسمحان بتخزين الأعداد العشرية (13.8، 16.911 ...).

سترى أنّنا نتعامل من الأعداد الصحيحة معظم الوقت لأنّها سهلة الإستخدام.

x احذر في الأعداد العشرية من استخدام الفاصلة، حاسوبك لا يستخدم سوى النقطة. لذلك لا تكتب 54,9 بدل 54.9 !

هذا ليس كلّ شيء، توجد أنواع أخرى تعرف بـ`unsigned` (عديمة الإشارة) تصلح لتخزين الأعداد الموجبة فقط.
يجب إضافة كلمة `unsigned` إلى النوع لاستخدامها.

`unsigned char`	من 0 إلى 255
`unsigned int`	من 0 إلى 65,535
`unsigned int`	من 0 إلى 4,294,967,295

كما ترى، مشكلة الأنواع عديمة الإشارة هي عدم القدرة على تخزين الأعداد السالبة، لكن الشيء الإيجابي هي أنّها توفّر لنا ضعف حجم التخزين لكلّ نوع موافق (مثلا `signed char` يتوقّف عند 127، بينما `unsigned char` يمتد إلى 255).

م تلاحظ أنّ النوع `char` قد تمّ إدراجه إمّا مع الكلمة المفتاحية `signed`، أو مع الكلمة المفتاحية `unsigned` لكن لا يوضع وحده أبدا. السبب بسيط : هذا النوع يمكن أن يكون بإشارة أو بدون إشارة حسب الحواسيب. لذلك أنصحكم بتحديد أي واحد منهما تريدون حسب نوع القيمة المراد تخزينها.

؟ لِمَاذا توجد ثلاثة أنواع من المتغيرات الصحيحة ؟ ألا يكفي نوع واحد ؟

بلى، و لكن إنشاء أنواع متعدّدة هدفه الاقتصاد من استهلاك الذاكرة. فعندما نطلب من الحاسوب حجز مساحة لمتغيّر من نوع `char` فهذا سيكون أقل من المساحة المستهلكة لو إخترنا متغيّرا من نوع `int`.

كان هذا مهمّا جدّا عندما كانت الحواسيب محدودة الذاكرة. أمّا اليوم فحواسيبنا بها ذاكرات كبيرة جدّا فلم يعد هذا يمثّل مشكلة. إذا احترت أيّ واحد من الأنواع تستخدم، فاختر `int` للأعداد الصحيحة (أو `double` للعشرية).

باختصار، نقوم بالتفريق خاصّة بين الأعداد الصحيحة و العشريّة

- بالنسبة للأعداد الصحيحة، نستعمل عادة `int`.
- بالنسبة للأعداد العشريّة نستعمل عادة `double`.

التصريح عن متغيّر

ها قد بدأنا. الآن سنقوم بإنشاء برنامج كونسول نسميه "variables".
سنتعلم كيف نصرّح عن متغيّر، أي الطلب من الحاسوب إذنا باستخدام شيء من ذاكرة الوصول العشوائي.

التصريح سيكون سهلا الآن، يجب علينا فقط أن نتّبع الترتيب التالي :

1. نحدّد نوع المتغيّر المراد إنشائه.
2. نترك فراغا.
3. نحدد اسم المتغيّر.
4. و أخيرا، يجب ألاّ ننسى أبدا وضع الفاصلة المنقوطة.

مثلا، إذا أردنا إنشاء المتغير `numberOfLives` من نوع `int` سنكتب السطر التالي :

```
int numberOfLives;
```

هذا كل ما في الأمر، وإليك بعض الأمثلة الغبيّة لملاحظة الشكل :

```
int mathMark;
double moneySumReceived;
unsigned int numberOfReadersWhoAreReadingALongVariableName;
```

حسنا، أعتقد أنّك فهمت المبدأ الآن !

ما فعلناه للتو يسمّى **تصريحا عن متغيّر** (Declaring a variable) (هذا مصطلح يجب حفظه). يمكنك وضع التصريحات في بدايات الدوال. و بما أن لدينا دالّة وحيدة فقط (الدالة `main`) فسنصرّح المتغيّر هكذا :

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) // Equivalent to int main()
{
  int numberOfLives;

  return 0;
}
```

إن قمت بتشغيل البرنامج فستلاحظ بدهشة ... أنّه لا يقوم بشيء.

بعض التوضيحات

في الواقع، هناك أشياء تحدث، لكنّك لا تراها. عندما يصل البرنامج إلى سطر التصريح عن المتغيّر فإنّه يطلب من الحاسوب بهدوء أن يعطيه شيئا من ذاكرة الوصول العشوائي.
إن تم كلّ شيء على أحسن حال (و هذا ما يقع في غالب الأحيان) فإن الحاسوب يوافق على هذا الطلب. المشكل الوحيد الذي يمكن أن يقع هو امتلاء الذاكرة، لكنّ هذا أمر نادر الحدوث، من الذي سيملؤ الذاكرة بمتغيرات `int` ؟

لذلك كن متيقناً أنه سوف يتم إنشاء المتغيرات بنجاح.

> م
> معلومة صغيرة : إن كان لديك عدد من المتغيّرات بنفس النوع تريد التصريح عنها، فمن غير الضروري كتابة سطر لكلّ متغيّر. يكفي فصل أسماء المتغيّرات عن بعضها بفواصل على نفس السطر، مثلا : `int numberOfLives, level, playerAge;` . هذا ينشئ ثلاثة متغيّرات أسماؤها على التوالي : `numberOfLives`، `level` و `playerAge`.

و الآن يجب علينا إعطاء قيمة للمتغيّر الذي أنشأناه.

إسناد قيمة إلى متغيّر

هذا أمر سهل للغاية فإن أردنا أن نعطي قيمة لمتغيرنا `numberOfLives` فيكننا ببساطة كتابة هذا :

```
1 numberOfLives = 5;
```

لن نفعل شيئا بعد هذا. ستضع اسم المتغير، إشارة تساوي، بعدها القيمة التي تريد أن تضعها بالداخل، في هذه الحالة سنعطي القيمة 5 للمتغير `numberOfLives` . برنامجنا الكامل سيكون هكذا :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) // Equivalent to int main()
5  {
6    int numberOfLives;
7    numberOfLives = 5;
8
9    return 0;
10 }
```

هنا أيضا لا شيء يُعرض على الشاشة، كلّ شيء يحدث في الذاكرة.
في أعماق الحاسوب هناك خانة ذاكرة تأخذ القيمة 5. أليس شيئا رائعا ؟

يمكننا أن نستمتع بتغيير القيمة :

```c
int numberOfLives;
numberOfLives = 5;
numberOfLives = 4;
numberOfLives = 3;
```

في هذا المثال، المتغير سيأخذ القيمة 5 ثم 4 ثم 3. و بما أنّ الحاسوب سريع جدّا ففي أقلّ من رمشة عين يأخذ المتغيّر القيم 5 ثمّ 4 ثمّ 3 ثمّ ينتهي البرنامج.

قيمة متغيّر جديد

هناك سؤال مهمّ يجب أن أطرحه عليك :

؟

عند التصريح عن متغيّر، أيّ قيمة تكون فيه ؟

عندما يقرأ الحاسوب هذا السطر :

```c
int numberOfLives;
```

فسيحجز مكانا في الذاكرة لهذا المتغيّر، لكن ما هي قيمته في هذه اللحظة ؟
هل يملك قيمة افتراضيّة ؟ (0 مثلا).

الجواب هو لا، لا و لا ثمّ لا ! لا توجد قيمة افتراضيّة. المكان محجوز لكنّ القيمة لا تتغيّر. لا يتمّ حذف ما يوجد في خانة الذاكرة. ستكون قيمة متغيّرك هي نفس القيمة التي كانت موجودة من قبل في تلك الخانة، و **يمكن أن تكون أيّ شيء !**

إن لم يتم تغيير هذا المكان من الذاكرة من قبل، فقد تكون قيمته 0، لكنّ هذا ليس مؤكّدا، قد يأخذ متغيّرك القيمة 363 أو 18، و هذا يدلّ على بقايا برنامج قد استخدم هذه الخانة من قبل.
و لهذا كي لا تعترضنا المشاكل لاحقا يجب تهيئة المتغير عند التصريح عنه، في لغة C هذا أمر ممكن حيث أنّ كل ما يجب القيام به هو الدمج بين التصريح و تعيين قيمة المتغير في نفس التعليمة كالتالي :

```c
int numberOfLives = 5;
```

هنا يتمّ التصريح عن المتغيّر ثمّ إسناد قيمة 5 له بطريقة مباشرة.
الشيء الإيجابي هنا أننا متأكدون من أنّ المتغيّر يحمل القيمة 5 مبدئيّا.

الثوابت

قد نريد أحيانا إنشاء متغيّر يحمل قيمة ثابتة طوال وقت تشغيل البرنامج. هذا يعني أنّه بمجرّد التصريح عنه، فإنّه يحافظ على قيمته و لا يمكن لأحد تغيير قيمته التي يحويها.

هذا النوع الخاص من المتغيّرات يسمى الثوابت (Constants)، طبعا هذا لأنّ قيمتها تبقى ثابتة.

للتصريح عن ثابت، تكفي إضافة كلمة `const` قبل نوع المتغيّر. و لكن يجب إعطاء الثابت قيمة بمجرّد التصريح عنه، لأنه لاحقا سيكون الوقت متأخّرا و لن تتمكّن من تغيير قيمته.

مثال على التصريح بثابت :

```
const int INITIAL_NUMBER_OF_LIVES = 5;
```

م

ليس من الضروري أن يكون اسم الثابت مكوّنا من حروف كبيرة فقط، لكنّ هذا يساعدنا على التفريق بينها و بين المتغيّرات. لاحظ أيضا أننا نستخدم الرمز _ بدلا من الفراغ.

بغضّ النظر عن هذا، الثابت يستعمل بشكل عادي كالمتغير ، يمكنك عرض قيمته إن أردت. الفرق الوحيد هو أنّه إذا حاولت تغيير قيمة الثابت في برنامجك فسينبّهك المترجم إلى وجود خطأ.

أخطاء الترجمة تعرض أسفل الشاشة (في المكان الذي أسميه "منطقة الموت" هل تتذكر ؟)، في هذه الحالة سيقوم المترجم بعرض رسالة تشبه التالي :

```
[Warning] assignment of read-only variable 'INITIAL_NUMBER_OF_LIVES'
```

3،4 عرض محتوى متغير

نحن نعرف كيف نعرض نصّا على الشاشة باستخدام الدالة `printf`.
الآن سنتعلم كيف نعرض القيمة الخاصة بالمتغير باستخدام نفس الدالة.

سنستخدم الدالة `printf` بنفس الطريقة، باستثناء أننا سنقوم بإضافة رمز خاص في المكان الذّي نريد عرض قيمة المتغيّر فيه. مثلا :

```
printf("You have %d lives left");
```

هذا الرمز الخاص ما هو إلّا `%` متبوعا بحرف (`'d'` في هذا المثال). هذا الحرف يبيّن ما الذي سنقوم بعرضه. `'d'` تعني أننا سنعرض قيمة متغيّر من نوع `int`.
توجد حروف أخرى عديدة، لكن من أجل التبسيط فسنكتفي بهذه :

الشكل	النوع المنتظر
%d	int
%ld	long
%f	float
%f	double

م لاحظ أنّ الشكل المستخدم لعرض `float` و `double` هو نفسه.

سأعلّمك رموزا أخرى في الوقت المناسب، حاليّا تذكّر هذه فقط.

شارفنا على الإنتهاء. حددنا موضع كتابة عدد صحيح، لكننا لم نذكر ما هو ! يجب علينا أن نحدد للدالة `printf` المتغيّر الذي نريد عرض قيمته.
لفعل ذلك، أكتب اسم المتغيّر بعد علامات الاقتباس بعد وضع فاصلة :

```
1 printf("You have %d lives left", numberOfLives);
```

`%d` سيتمّ استبداله بقيمة المتغيّر المكتوب بعد الفاصلة، أي `numberOfLives`.
فلنجرّب هذا في برنامج :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6    int numberOfLives = 5; // The player has 5 lives in the beginning.
7
8    printf("You have %d lives left\n", numberOfLives);
9    printf("**** B A M ****\n"); // A big blow on his head.
10   numberOfLives = 4; // Life lost
11   printf("Sorry, only %d lives are remaining now !\n\n", numberOfLives);
12
13   return 0;
14 }
```

يمكن أن يكون البرنامج قاعدة للعبة فيديو (ينقصه الخيال فقط).
هذا البرنامج يعرض على الشاشة :

```
You have 5 lives left
**** B A M ****
Sorry, only 4 lives are remaining now !
```

يجب أن تفهم ما الذي يحدث في البرنامج :

1. في البداية يملك اللاعب 5 أرواح، نعرض هذا في `printf`.
2. بعدها يتلقى اللاعب ضربة على رأسه (في مكان BAM).
3. بعدها لا يبقى له سوى 4 أرواح، نعرض هذا أيضا باستخدام `printf`.

كان ذلك سهلا !

عرض عدّة متغيّرات باستخدام printf واحدة

يمكن عرض قيم متغيرات عديدة بنفس الدالة `printf` . يكفي فقط أن تضع `%d` أو `%f` في الأمكنة المناسبة، ثم تحديد المتغيّرات الموافقة بنفس الترتيب، مفصولة عن بعضها البعض بفواصل.

مثال :

```
printf("You have %d lives and you are in the level n° %d", numberOfLives, level
    );
```

! يجب عليك تحديد المتغيّرات بالترتيب الصحيح، `%d` الأولى توافق المتغيّر الأوّل (`numberOfLives`) و `%d` الثانية توافق المتغيّر الثاني (`level`). إذا أخطأت في الترتيب، فلن يكون للجملة أيّ معنى.

حسنا، فلنقم بتجربة صغيرة الآن. لاحظ أنني قد حذفت توجيهات المعالج القبلي (أعني تلك السطور التي تبدأ برمز `#`)، أنا أفترض أنك تضعها في كلّ مرّة الآن :

```
int main(int argc, char *argv[])
{
  int numberOfLives = 5, level = 1;

  printf("You have %d lives and you are in the level n° %d", numberOfLives,
      level);

  return 0;
}
```

و هذا يعرض لك :

```
You have 5 lives and you are in the level n° 1
```

4.4 استرجاع إدخال

بدأت المتغيرات تصبح ممتعة الآن. سنتعلم كيف نطلب من المستخدم كتابة عدد في الكونسول. سنسترجع هذا العدد و نحفظه في متغير. عندما نفعل ذلك سيكون بإمكاننا القيام بكثير من الأمور، سوف ترى ذلك.

لكي نطلب من المستخدم إدخال شيء سنقوم باستخدام دالة جاهزة تسمّى : `scanf` .
هذه الدالة تشبه إلى حد كبير `printf` . يجب عليك وضع شكل لتحديد ما سيدخله المستخدم (`int`، `float` ...). ثم يجب عليك تعيين إسم المتغير الذي سيأخذ هذا العدد.

هذا ما سنفعله على سبيل المثال :

```
int age = 0;
scanf("%d", &age);
```

علينا وضع `%d` بين مزدوجتين. كما يجب وضع `&` أمام اسم المتغيّر الذي سيستقبل القيمة.

؟

و لكن لماذا نضع `&` أمام اسم المتغيّر ؟

هنا يجب عليك أن تثق بي. لأنه لا يمكنني أن أشرح لك ما الذي تعنيه في الوقت الحالي. لكني أضمن لك أنّي سأقوم بشرحه في وقت لاحق.

X

احذر ! يوجد اختلاف صغير في الشكل بين `printf` و `scanf` ! لاسترجاع `float` يجب استخدام الشكل `"%f"` ، أما من أجل النوع `double` فيتم استخدام `"%lf"` .

```
double weight = 0;
scanf("%lf", &weight);
```

فلنعد إلى برنامجنا. عندما يصل هذا الأخير إلى `scanf` ، يتوقف مؤقّتا منتظرا المستخدم لكي يدخل عددا. هذا العدد يتم حفظه في المتغير `age` . إليك برنامجا يطلب عمر المستخدم ثم يقوم بعرضه :

```
int main(int argc, char ▫ argv[])
{
  int age = 0;

  printf("How old are you ? ");
  scanf("%d", &age);
  printf("Ah ! so you are %d years old !\n\n", age);

  return 0;
}
```

و هذا ما يعرضه :

```
How old are you ? 20
Ah ! so you are 20 years old !
```

يتوقف البرنامج مؤقّتا بعد عرض السؤال و يظهر مؤشرا على الشاشة، عليك إذن أن تكتب عددا (عمرك). إضغط بعدها على المفتاح "إدخال" (Enter). بعد ذلك يواصل البرنامج عمله. هنا، يقوم البرنامج بعرض قيمة المتغيّر `age` على الشاشة.

حسنا، لقد فهمت الأساس. بفضل الدالة `scanf` يمكننا البدء في التفاعل مع المستخدم.

ليكن في علمك أنه لا مانع من إدخال أي شيء غير عدد صحيح :

• إن قمت بإدخال عدد عشري، مثلا 2،9، فسيتم قطعه تلقائيّا. في هذه الحالة سيتم حفظ العدد 2 في المتغيّر.

- إن قمت بكتابة أحرف عشوائية، مثلا : éèdyf، فلن تتغيّر قيمة المتغيّر. و الشيء الجيّد في هذه الحالة هو أننا قمنا بتهيئة قيمة المتغير على 0. لذلك سيعرض البرنامج : ”Ah ! so you are 0 years old“ إن لم يتم الأمر بشكل صحيح. لو لم نقم بتهيئة المتغيّر فسيعرض البرنامج عددا عشوائيا !

ملخّص

- حواسيبنا تملك عدة أنواع من الذاكرة. من الأسرع إلى الأبطأ : السجلّات، ذاكرة التخبئة، الذاكرة الحيّة و القرص الصلب.
- للاحتفاظ بالمعلومات، يحتاج برنامجنا إلى تخزين البيانات في الذاكرة. لهذا يستخدم **الذاكرة الحية**. السجلّات و ذاكرة التخبئة تستخدم أيضا لزيادة الأداء، لكنّ هذا يحدث تلقائيا و ليس علينا أن نهتم بهذا الأمر.
- في الشفرة المصدرية، **المتغيّرات** هي البيانات المحفوظة مؤقّتا في الذاكرة الحيّة. قيمة هذه البيانات يمكن أن تتغيّر أثناء تشغيل البرنامج.
- بالمقابل، نطلق اسم **الثوابت** على بيانات محفوظة في الذاكرة الحيّة. قيمة هذه البيانات لا يمكن أن تتغيّر.
- توجد أنواع عديدة من المتغيّرات، تختلف في حجم الذاكرة التي تشغله. بعض الأنواع مثل `int` تستخدم لتخزين عدد صحيح، و أخرى مثل `double` تخزّن أعدادا عشريّة.
- الدالة `scanf` تمكّننا من طلب عدد من المستخدم.

الفصل 5

حسابات سهلة

كما قلت لك في الفصل السابق : جهازك ماهو إلا آلة حاسبة كبيرة. سواء كنت تسمع الموسيقى، تشاهد فلماً أو تلعب لعبة، فإن الحاسوب ينجز الحسابات طيلة الوقت.

هذا الفصل سيساعدك على التعرف على معظم الحسابات التي يقوم بها الجهاز. سنعيد استعمال ما نحن بصدد تعلّمه عن عالم المتغيرات. الفكرة هي أننا سنقوم بعمليات على المتغيرات : نجمعها، نضربها، نخزّن النتائج في متغيرات أخرى، إلخ.

حتى و إن لم تكن من هواة الرياضيات، فإن هذا الفصل إلزامي و لا مفرّ منه.

1.5 الحسابات القاعدية

بالرغم من قدرة الجهاز الواسعة إلا أنه في الأساس يتعمد في حساباته على عمليات بسيطة للغاية و هي :

- الجمع،
- الطرح،
- القسمة،
- الضرب،
- الترديد (Modulo) (سأشرح لاحقا ما الّذي يعنيه إذا لم تكن تعرفه الآن).

إن كان بودك القيام بحسابات أكثر تعقيدا (كالأسس و اللوغاريثم و ماشابه)، يجب عليك إذا برمجتها أو بمعنى آخر : **توضح للجهاز كيف يقوم بها.**
لحسن الحظ، سترى لاحقاً في هذا الفصل أنه توجد مكتبة في لغة الـC، تحتوي على دوال رياضية جاهزة. لن يكون عليك إعادة كتابتها إلا إذا أردت فعل ذلك تطوّعيا أو كنت أستاذ رياضيات.

لنبدأ بالعملية الأسهل و هي الجمع.
طبعا في الجمع نحتاج الرمز +.
يجب وضع نتيجة الجمع في متغير و لهذا سنقوم بإنشاء متغير اسمه مثلا `result` من نوع `int` و نقوم بالحساب :

```
1  int result = 0;
2  result = 5 + 3;
```

لا يجب أن تكون محترفا في الحساب الذهني لتعرف أن النتيجة ستكون 8 بعد تشغيل البرنامج.
بالطبع البرنامج لن يظهر أية نتيجة باستعمال هذه الشفرة المصدرية. إذا أردت معرفة محتوى المتغير `result` عليك باستعمال الدالة `printf` التي تجيد كيفية استخدامها جيداً الآن :

```
1  printf("5 + 3 =  %d", result);
```

و هذا ما سيظهر على الشاشة :

```
5 + 3 = 8
```

و هكذا نهي عملية الجمع بسهولة.
الأمر مماثل بالنسبة للعمليات الأخرى، نحتاج تغيير الرمز ليس إلا :

العمليّة	الرمز
الجمع	+
الطرح	-
الضرب	*
القسمة	/
الترديد	%

إذا كنت قد استعملت من قبل الآلة الحاسبة الخاصة بحاسوبك، فيفترض بك أن تكون متعوّدا على هذه الإشارات.
لا يوجد أي شيء صعب بخصوصها باستثناء القسمة و الترديد اللذان سأشرحهما فيما يلي بالتفصيل.

القسمة

ينجز الحاسوب عملية القسمة بشكل طبيعيّ عندما لا يوجد أي باق. مثلا العملية `6 / 2` تعطينا النتيجة 3، النتيجة صحيحة. حتّى الآن، لا مشكلة.

لكن لو نأخذ الآن عملية قسمة بباقٍ مثل `5 / 2` ... نتوقع أن النتيجة ستكون 2.5، و لكن أنظر إلى ما تعطيه الشفرة :

```
1  int result = 0;
2  result = 5 / 2;
3  printf("5 / 2 =  %d", result);
```

```
5 / 2 = 2
```

هناك مشكل كبير، فنحن نتوقّع أن نحصل على القيمة 2.5، لكن الحاسوب أعطى القيمة 2 !

هل يا ترى أجهزتنا غبية لهذه الدرجة ؟
في الواقع، يقوم الجهاز بعملية قسمة صحيحة (إقليدية) أي أنه يحتفظ بالجزء الصحيح فقط الذي هو 2.

> ؟ ها أنا أعرف السبب ! لأن المتغير `result` الذي استخدمناه هو من نوع `int` ! لو استخدمنا النوع `double` لاستطاع تخزين العدد العشري !

لا، ليس هذا هو السبب ! جرب نفس الشفرة بتغيير نوع النتيجة إلى `double` و ستجد بأننا نحصّل على نفس النتيجة 2 لأن طرفا العملية من نوع `int` فإن الحاسوب سيعيد نتيجة من نوع `int`.

إن أردنا أن يظهر لنا الجهاز القيمة الصحيحة، يجب أن نغير العددين 2 و 5 إلى عددين عشريين كالتالي : 2.0 و 5.0 (قيمتهما هي نفسها لكن الجهاز سيعتبرهما عددين عشريين، و بالتالي هو يظن بأنه يقوم بقسمة عددين عشريين) :

```c
double result = 0;
result = 5.0 / 2.0;
printf("5 / 2 =  %f", result);
```

```
5 / 2 = 2.500000
```

هنا العدد صحيح بالرغم من وجود عدة أصفار في نهاية العدد، لكنّ القيمة تبقى نفسها.

فكرة القسمة الإقليدية التي يقوم بها الحاسوب مهمة، تذكّر أنه بالنسبة للحاسوب :

- $5/2 = 2$،
- $10/3 = 3$،
- $4/5 = 0$.

هذا مفاجئ بعض الشيء، لكنّها طريقته في التعامل مع الأعداد الصحيحة.

إن أردت الحصول على نتيجة عشريّة، فيجب أن يكون حدّا العملية عشريّين :

- $5.0/2.0 = 2.5$،
- $10.0/3.0 = 3.33333$،
- $4.0/5.0 = 0.8$.

يمكن القول أن الجهاز يطرح على نفسه السؤال : "كم يوجد من 2 في العدد 5 ؟" طبعا يوجد 2 فقط.

و لكن أين الباقي من العملية ؟ لأنني لما أقول 5 هي أثنين من 2 ، يبقى 1 طبعا، كيف لنا أن نسترجعه ؟ هنا يتدخل الترديد الذي كلمتك عنه.

الترديد

هو عبارة عن عملية حسابية تسمح بالحصول على باقي عملية القسمة، و هي عملية غير معروفة مقارنة بالعمليات الأربع الأخرى، لكن الجهاز يعتبرها من العمليات القاعدية، و يمكن اعتبارها حلا لمشكل قسمة الأعداد الطبيعية.

كما قلت لك الترديد يمثل بالرمز `%`.
إليكم بعض الأمثلة :

- $5\%2 = 1$،
- $14\%3 = 2$،
- $4\%2 = 0$.

الترديد `5 % 2` هو باقي العملية `5 / 2` مما يعني أن الجهاز يقوم بالعملية `5 = 2 * 2 + 1` حيث أن 1 هو الباقي و الذي يقوم بإرجاعه الترديد.

نفس الشيء بالنسبة للعملية `14 % 3`، العملية هي `14 = 3 * 4 + 2` (الترديد يعطي القيمة 2). أخيرا، من أجل `4 % 2`، القسمة تامة، فلا يوجد باقي، لهذا يعطي الترديد القيمة 0.

حسنا، لا يوجد ما يمكنني إضافته بخصوص عملية الترديد. كان هذا فقط شرحا لمن لا يعرفها.

لدي خبر جيد آخر، و هو أننا أتممنا كلّ عمليات الحساب القاعدية و تخلصنا من درس الرياضيات !

عمليات على المتغيرات

الشيء الجيد هو أنه بعد أن تعلمت كيف تستخدم العمليات القاعدية، يمكنك الآن أن نتعلّم كيفية القيام بهذه العمليات على المتغيرات.
لا شيء يمكنه منعك من كتابة الشفرة التالية :

```
result = number1 + number2;
```

هذا السطر يعمل على جمع المتغيرين `number1` و `number2` ثم يخزن النتيجة في المتغير `result`.

هنا بدأت الامور الممتعة تظهر، و حقيقة، مستواك الحالي يسمح لك ببرمجة آلة حاسبة بسيطة. نعم، نعم، أؤكّد لك ذلك !

تخيل وجود برنامج يطلب من المستخدم إدخال عددين، ثم يقوم بتخزينهما في متغيرين، ثم يجمع هذين المتغيرين و يخزن النتيجة في متغير اسمه `result`. لم يبق سوى إظهار النتيجة على الشاشة في وقت لا يتمكّن فيه المستخدم حتى من تخمين النتيجة.

حاول كتابة هذا البرنامج البسيط، إنّه سهل و سيكون تدريبا لك !

إليك الجواب :

```c
int main(int argc, char * argv[])
{
  int result = 0, number1 = 0, number2 = 0;

  // We request the two numbers from the user :

  printf("Enter the first number : ");
  scanf("%d", &number1);
  printf("Enter the second number : ");
  scanf("%d", &number2);

  // We calculate the result :

  result = number1 + number2;

  // We display the result on the screen :

  printf("%d + %d = %d\n", number1, number2, result);

  return 0;
}
```

```
Enter the first number : 30
Enter the second number : 25
30 + 25 = 55
```

بدون أن تشعر، لقد أنشأت أول برنامج لك ذو فائدة. إنّه قادر على جمع عددين و إظهارا النتيجة على الشاشة !

يمكنك التجريب باستخدام أعداد أخرى (يجب ألا تجاوز الحد الأقصى لتحمّل نوع الـ `int`) و سيقوم الحاسوب بالحساب بشكل سريع جداً لا يتجاوز بعض أجزاء من المليار من الثانية !

أنصحك أيضاً بتجريب العمليات الأخرى (الطرح، القسمة و الضرب) لكي تتدرب. لن يكون هذا متعبا إلّا بقدر تغيير إشارة أو ثنتين. يمكنك أيضاً إضافة متغير ثالث و جمع ثلاثة متغيرات دفعة واحدة. سيشتغل البرنامج دون مشاكل :

```c
result = number1 + number2 + number3;
```

2.5 الاختصارات

كما وعدتك، لا توجد عمليات أخرى لنتعلّمها اليوم لأن هذه هي كلّ العمليات الموجودة ! بهذه العمليات البسيطة يمكنك برمجة أي شيء تريده. أعلم أنّه يصعب عليك التصديق لو قلت لك أن لعبة ثلاثية الأبعاد ليست في النهاية سوى مجموعة من عمليات الجمع و الطرح ... لكنها الحقيقة.

توجد طرق في لغة الـC تسمح لنا باختصار كتابة بعض العمليات. لماذا نستعمل هذه الإختصارات ؟ لأننا نحتاج في غالب الأحيان من كتابة عمليات مكرّرة. ستفهم ما أريد قوله حينما ترون ما نسمّيه بالزيادة.

الزيادة (Incrementation)

في غالب الأحيان ستضطر إلى إضافة 1 إلى محتوى متغير. و بالتقدّم في برنامجك، تكون لديك متغيرات يزيد محتواها في كلّ مرة بـ1.

نفترض أن لديك متغيرا يحمل اسم `number`، هل تعرف كيف تضيف له 1 دون أن تعرف محتواه ؟

إليك ما يجب عليك فعله :

```
1 number = number + 1;
```

ما الذي يحصل هنا ؟ نقوم بالحساب `number + 1` ثم نخزن الناتج في المتغير `number` ! و منه فإن كان المتغير يحمل القيمة 4 فهو بعد العملية يحمل القيمة 5. لو أنه كان يحمل القيمة 8، فهو الآن يحمل القيمة 9، إلخ.

هذه العملية تتكرر كثيرا. و بما أن المبرمج شخص كسول، سيتعبه أمر كتابة اسم المتغير مرتين في نفس التعليمة (نعم هذا أمر متعب !). لهذا تم اختراع اختصار لهذه العملية بما نسميه بـ**الزيادة** (Incrementation) التعليمة أسفله تعطي تماما نفس نتيجة التعليمة السابقة :

```
1 number++;
```

هذا السطر له نفس وظيفة السطر السابق الذي كتبناه قبل قليل، أليس مختصرا و قصيرا ؟ إنه يعني "إضافة 1 لمتغير". يكفي إذا أن نرفق باسم المتغير `number` الاشارة + مرتين، مع عدم نسيان الفاصلة المنقوطة الخاصة بنهاية التعليمة.

هذه العملية ستساعدنا كثيرا مستقبلا لأننا سنضطر للقيام بعملية الزيادة كثيرا.

> م
> إذا كنت دقيق الملاحظة، كنت لتلحظ أن إشارتي ++ متواجدتان أيضاً في اسم اللغة ++C. أنت الآن قادر على أن تفهم السر وراء ذلك ! الـ++C تعني أننا نتكلم عن لغة الـC "مع زيادة". عمليّا، يسمح لنا الـ++C بالبرمجة بطريقة مختلفة، لكن لا يعني أنه "أفضل" من الـC. هو فقط مختلف.

الإنقاص (Decrementation)

إنّها بكلّ بساطة عكس عملية الزيادة، فهي تقوم بإنقاص 1 من متغير.
بالرغم من أن عملية الزيادة هي أكثر استعمالاً إلا أن عملية الإنقاص تبقى شائعة أيضا.

إليكم كيف ننقص 1 من متغير بالشكل "الطويل" :

```
1 number = number — 1;
```

و في شكلها المختصر :

```
1 number—;
```

ربّما كان بإمكانك تخمين ذلك وحدك ! بدل وضع إشارة `++` نضع إشارة `--`. إذا كان محتوى المتغير هو 5 فسيصبح بعد الإنقاص يساوي 4.

الاختصارات الأخرى

توجد اختصارات أخرى تعمل بنفس المنطق. هذه الاختصارات تصلح لكل العمليات القاعدية : `+` `-` `*` `/` `%`.
هي تساعدنا على تجنب تكرار نفس اسم المتغير في نفس التعليمة.
لضرب محتوى المتغير في 2 مثلا نقوم بالتالي :

```
1  number = number * 2;
```

و بالشكل المختصر :

```
1  number *= 2;
```

بالطبع إن كان للمتغير القيمة 5 قبل إجراء العملية فسيحمل الآن 10 بعد هذه التعليمة.
بالنسبة لباقي العمليات القاعدية، فالمبدأ نفسه. إليك برنامجا صغيرا كمثال :

```
1  int number = 2;
2  number += 4; // number = 6 ...
3  number -= 3; // ... number = 3
4  number *= 5; // ... number = 15
5  number /= 3; // ... number = 5
6  number %= 3; // ... number = 2 (because 5 = 1 * 3 + 2)
```

(لا تتذمر فبعض الحسابات الذهنيّة لن تقتل شخصاً !)

الشيء الجيد هنا أنه يمكننا استعمال اختصارات على كلّ العمليات القاعدية، فيمكننا أن نجمع، نطرح، نضرب أيّ عدد.
هي اختصارات عليك تعلّمها إن كان البرنامج الذي تكتبه يحتوي الكثير من التعليمات المكرّرة.

تذكّر أيضاً أن عملية الزيادة هي الاختصار الأكثر استعمالاً.

3.5 المكتبة الرياضياتيّة

في لغة الـC هناك دائما ما نسميه بالمكتبات القياسية (Standard libraries)، و هي المكتبات التي تستخدم على الدوام.
إنّها مكتبات قاعدية تستخدم كثيرا.

أذكرك بما قلت سابقا، المكتبة هي مجموعة دوال جاهزة. هذه الدوال تمت كتابتها من طرف مبرمجين قبلك، و هي تساعدك على تجنب إعادة اختراع العجلة في كلّ برنامج جديد.

في الواقع، العمليات الخمس القاعدية التّي رأيناها هي أقلّ من أن تكون كافية. إذا لم تفهم هذا، فربّما قد تكون صغيرا في السنّ أو لم تتعلّم الكثير عن الرياضيّات في حياتك. المكتبة الرياضياتيّة تحوي العديد من الدوال الأخرى التّي قد تحتاجها.

أعطيك مثالا، لغة الـC لا تحتوي على عملية الأس ! كيف نحسب المربع ؟ يمكنك كتابة العملية 52 في برنامجك لكن الجهاز لن يفهمها أبداً لأنّه لا يعرف مالّذي تعنيه هذه، إلا إن قمت بشرح العملية له باستخدام المكتبة الرياضياتيّة !

يمكننا الاستعانة بالدوال الجاهزة في المكتبة الرياضياتيّة، لكن لا تنس كتابة توجيهات المعالج القبلي الخاصة بها في بداية كل برنامج :

```c
#include <math.h>
```

ما إن تكتب السطر السابق حتّى تصبح قادراً على استخدام كل الدوال المتوفرة في هذه المكتبة.

لديّ نيّة في عرضها لك الآن.
حسنا، بما أنّه يوجد الكثير منها، فلا يمكنني إنشاء قائمة كاملة هنا. من جهة لأنّ هذا سيكون كثيرا للفهم، و من ناحية أخرى، فأصابعي المسكينة ستذوب قبل إنهاء كتّابة هذا الفصل ! سأريك إذن الدوال الرئيسيّة فقط، أي الّتي تبدو أكثر أهميّة.

> م
> ربّما قد لا يكون لديك مستوى الرياضيات اللازم لفهم ما تفعله هذه الدوال. إن كان هذا هو حالك، فلا تقلق. قم بالقراءة فقط، فهذا لن يضرّك فما يلي.
> على الرغم من ذلك، أقدّم لك نصيحة مجّانيّة : كن منتبها في دروس الرياضيات، لا نقول هذا من دون سبب، هذا سيفيدك في النهاية.

fabs

هذه الدالة تحسب القيمة المطلقة للرقم، و نرمز لها بالشكل التالي : |x|.
القيمة المطلقة لعدد هو قيمته الموجبة :

- إعطاء العدد 52− للدالة يجعلها ترجع القيمة 52،
- إعطاء العدد 52 للدالة يجعلها تعيد 52.

باختصار، تعيد دائما العدد الموجب الموافق لما أعطيته لها.

```c
double absolute = 0, number = -27;
absolute = fabs(number); // absolute = 27
```

هذه الدالّة تعيد `double`، لذا فالمتغيّر `absolute` يجب أن يكون من نوع `double`.

> م
> هناك دالة اخرى مماثلة لـ `fabs` تسمى `abs`، نجدها في `stdlib.h`.
> إنّها تعمل بنفس طريقة الأولى إلا أنها تعمل مع الأعداد الصحيحة، فهي تعيد `int`.

ceil

هذه الدالّة تعطي أول عدد طبيعي بعد العدد العشري الذي نعطيه لها. يمكن القول أنها تدوّر العدد دائما إلى العدد الذي يعلوه في الجزء الصحيح.
لو نعطيها مثلا العدد 26.512 فستعطينا العدد 27.

الدالة تعمل بنفس الطريقة و تعيد `double` أيضا :

```c
double above = 0, number = 52.71;
above = ceil(number); // Above = 53
```

floor

هذه عكس السابقة، تعيد العدد الأقل مباشرة في الجزء الصحيح.
إذا أعطيتها مثلا العدد 37.91 تعطيني العدد 37، يعني الجزء الصحيح.

pow

هذه خاصة بحساب قوى عدد (الأسس). يجب أن تعطيها قيمتين، الأولى هي العدد الذي تريد إجراء العملية عليه و الثانية هي القوة الّتي يجب رفع العدد إليها. هذا مخطط الدالة :

```
pow(number, power);
```

كمثال : "2 قوة 3" (الّتي نكتبها عادة 23 على الحاسوب) هو الحساب $2 * 2 * 2$ الّذي يعطي النتيجة 8 :

```
double result = 0, number = 2;
result = pow(number, 3); // result = 2^3 = 8
```

sqrt

هذه الدالة تحسب الجذر التربيعي لعدد، تعيد `double`.

```
double result = 0, number = 100;
result = sqrt(number); // Result = 10
```

sin، cos، tan

إنّها الدوال المثلثية الثلاث الشهيرة.
طريقة عملها هي نفسها، تعيد `double`.

هذه الدوال تأخذ قيما بـالراديان (Radians).

asin، acos، atan

و هي الدوال arcsin، arccos، arctan، دوال مثلثية أخرى.
تُستخدم بنفس الطريقة و تعيد `double` أيضا.

exp

هذه الدالة تحسب قيمة الدالة الأسيّة ذات الأساس e لعدد معين.
تعيد `double`.

log

هذه الدالة تحسب اللوغاريتم النيبيري لعدد معين. (الّذي نرمز له أيضا بـ"ln")

log10

هذه الدالة تحسب اللوغاريتم ذو الأساس 10 لعدد.

ملخّص

- الحاسوب ما هو سوى **آلة حاسبة كبيرة** : كل ما يجيد فعله هو القيام بالعمليّات.
- العمليات التي يجيدها الحاسوب **قاعدية جدا** : الضرب، القسمة، الجمع، الطرح و الترديد (باقي القسمة).
- بالإمكان **إجراء عمليات على المتغيرات**، الحاسوب سريع جداً في هذا النوع من العمليات.
- **الزيادة** (Incrementation) هي عملية إضافة الرقم 1 إلى متغير. نكتبها `variable++`.
- **الإنقاص** (Decrementation) هي عملية طرح الرقم 1 من متغير. نكتبها `variable--`.
- لزيادة عدد العمليات التي يمكن للحاسوب القيام بها، نستعمل **المكتبة الرياضيّاتيّة** (أي `#include <math.h>`)
- تحتوي هذه المكتبة على **دوال رياضيّاتيّة متقدّمة** كالأسّ و الجذر و اللوغاريثم و غيرها.

الفصل 6

الشروط (Conditions)

لقد رأينا فيما سبق بأن هناك العديد من لغات البرمجة. بعضها متشابه : الكثير منها مستلهم من الـC.

في الواقع، لغة الـC أُنشئت منذ زمن طويل، و هذا جعلها نموذجا للّغات الجديدة.

لغات البرمجة تختلف في بعض الأمور، لكن هناك مبادئ لا يمكن أن تخلو منها أية لغة برمجية. لقد رأينا كيف ننشئ المتغيّرات، كيف نقوم بالحسابات، و الآن سنمرّ إلى **الشروط**.
من دون استعمال الشروط، برامجنا ستقوم دائما بنفس العمل !

1.6 الشرط `if ... else`

الشروط تسمح لنا بأن نقوم باختبارات على المتغيرات. مثلا يمكننا القول، "إن كان المتغيّر `machine` يساوي 50، يجب أن نقوم بكذا و كذا. و لكن من المؤسف عدم إمكانية اختبار سوى المساواة ! يجب أيضا اختبار ما إن كان المتغيّر، أقل من 50، أقل أو يساوي 50، أكبر، أكبر أو يساوي ... لا تقلق، في الـC كل شيء معَد !

لدراسة الشرط `if ... else` يجب أن نتبع المخطط التالي :

- نتعلم بعض الرموز قبل البدأ،
- الشرط `if`،
- الشرط `else`،
- الشرط `else if`،
- كثير من الشروط في مرة واحدة،
- بعض الأخطاء لنتجنبها.

قبل أن نرى كيف نكتب شرطا من النوع `if ... else` في الـC، يجب أن تعرف ثلاثة رموز أساسية. هذه الرموز ضرورية لإنشاء الشروط.

بعض الرموز للتعلّم

الجدول التالي يحوي رموز لغة الـC الّتي **يجب حفظها عن ظهر قلب** :

الرمز	المعنى
==	يساوي
>	أكبر
<	أصغر
<=	أصغر أو يساوي
>=	أكبر أو يساوي
!=	لا يساوي

X

انتبه جيدا، هناك رمزا مساواة `==` لنقوم باختبار المساواة. فالمبتدؤون يقومون غالبا باقتراف خطأ وضع إشارة واحدة `=` ، و هذا لديه معنى مختلف في الـC. سأذكّرك بهذا لاحقا.

if بسيط

فلنبدأ، لنقم باختبار بسيط، يقول للحاسوب : إن كان المتغير يساوي كذا فلنقم بكذا.

في الإنجليزيّة، كلمة "إذا" تُترَجم إلى `if` . و هذا ما الّذي نستخدمه في لغة الـC لإنشاء اختبار. أكتب إذن `if` ثم الأقواس و في داخلها الشرط.

بعد ذلك افتح حاضنة `{` و اغلقها لاحقا `}` . كلّ ما يوجد بين الحاضنتين سيتمّ تشغيله فقط في حالة تحقق الشرط.

هذا يعطينا إذن :

```
if (/* Your condition */)
{
	// Instructions to be executed
}
```

في مكان التعليق "Your condition" سنكتب شرطا للتحقّق من متغيّر.
كمثال سنحاول أن نقوم بختبار متغيّر `age` لاحتواء العمر. سنختبر ما إن كان المستعمل راشدا أم قاصرا استنادا إلى إذا **ما كان عمره يساوي أو أكبر من 18** :

```
if (age >= 18)
{
	printf("You are major !");
}
```

`>=` يعني "أكبر أو يساوي"، كما رأينا في الجدول السابق.

> م
> عندما لا يكون هناك سوى تعليمة واحدة بين الحاضنتين، يمكننا أن نتخلى عنهما. لكنني أفضل أن تضعهما دائما من أجل قراءة أفضل للشفرة.

جرّب هذه الشفرة

لكي نجرّب الشفرات السابقة و نفهم كيف يعمل الـ `if`، يجب أن نضعه داخل الدالة `main` و لا ننس التصريح بالمتغير `age` و إعطاءه قيمة ابتدائية.

هذا قد يبدو بديهيّا للبعض، و لكنّ كثيرا من القرّاء قد ضاعوا في هذه الأسطر و هذا ما دفعني لإضافة هذا الشرح.
هذه شفرة كاملة يمكنك تجريبها :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char *argv[])
4  {
5          int age = 20;
6
7          if (age >= 18)
8          {
9                  printf("You are major !\n");
10         }
11         return 0;
12 }
```

هنا، `age` يساوي 20 إذن سيتم عرض "You are major !".
جرّب تغيير القيمة إلى 15 مثلا، سيصبح الشرط خاطئاً و بالتالي لن يُعرض شيء هذه المرّة.

استخدم هذه الشفرة لتجريب الأمثلة اللاحقة في هذا الفصل.

مسألة نظافة

الطريقة الّتي تفتح بها الحاضنات ليست مهمّة. سيعمل برنامجك إن كتبت كلّ شيء على نفس السطر. مثال :

```
1  if (age >= 18) { printf("You are major !"); }
```

و على الرغم من أنّه ممكن، فهذا أمر **غير منصوح به مطلقا.**
في الواقع، الكتابة على نفس السطر تجعل شفرتك صعبة القراءة. إذا لم تتعوّد من الآن على تهوية شفرتك، فلاحقا عندما تكتب شفرات كبيرة ستضيع بالتأكيد !

حاول عرض شفرتك بنفس طريقتي : حاضنة على سطر، ثمّ التعليمات (مسبوقة بجدولة (tabulation))، ثمّ حاضنة الإغلاق على سطر آخر.

> م
> توجد طرق عديدة لعرض الشفرة بشكل جيّد. هذا لا يغيّر في عمل البرنامج شيئا، لكنّها مسألة "نمط معلوماتيّ" إن أردت. إذا رأيت الشفرة شخص آخر معروضة بشكل مختلف قليلا، فهذا لأنّه يبرمج بنمط مختلف. الهدف من هذا كلّه هو أن تبقى الشفرة مهوّاة و مقروءة.

else لكي نقول : و إلا

الآن و بما أنّك تعرف كيف تقوم باختبار بسيط، سنذهب بعيدا : إن لم يعمل الشرط (كان خاطئا)، فسنطلب من الحاسوب تشغيل تعليمات أخرى.

هذا يشبه ما يلي : إن كان المتغير يساوي كذا فلنقم بكذا، و إلا فلنقم بكذا.

يكفي أن نضيف `else` بعد الحاضنة الأخيرة لأوامر الـ `if`، مثلا :

```
1 if (age >= 18)
2 {
3         printf("You are major !");
4 }
5 else {
6         printf("You are minor !");
7 }
```

الأمر سهل : إن كان المتغيّر `age` أكبر من أو يساوي 18 سنكتب على الشاشة "You are major !"، و إن لم يكن الأمر كذلك فسنكتب "You are minor !".

if else لكي نقول : و إلا فإذا

سنرى كيف نقوم بـ"إذا" و "إلّا". يمكن أيضاً القيام بـ"و إلاّ فإذا" للقيام باختبار آخر في حالة ما إذا لم ينجح الأوّل. هذا الاختبار يوضع بين `if` و `else`.

يمكن ترجمة ذلك بما يلي : إن كان المتغير يساوي كذا، فافعل كذا و إلا فإن كان يساوي كذا، فافعل كذا و إلا (جميع الحالات المتبقية)، افعل كذا.

الترجمة بلغة C :

```
1 if (age >= 18)
2 {
3         printf("You are major !");
4 }
```

```c
5  else if (age > 4)
6  {
7          printf("Well you're not too young anyway...");
8  }
9  else
10 {
11         printf("Aga gaa aga gaaa");
12         // Baby language, you can't understand !
13 }
```

الجهاز يقوم بالاختبارات التالية بالترتيب :

- سيختبر الـ `if` الأول، إن كان الشرط محققا، فسيقوم بتنفيذ التعليمات الموجودة بين الحاضنتين الأولتين.
- إن لم يكن الشرط الأول محققا (يعني أننا في الـ `else if`) سيختبر الشرط الثاني، إن كان هذا الأخير محققا فإنه سينفذ التعليمات الموجودة بين الحاضنتين الثانيتين.
- في حالة ما لم يكن الشرط الأول محققا و لا حتى الثاني، فإنه سينفذ التعليمات الموجودة بين الحاضنتين الأخيرتين.

> م
> الـ `else` و الـ `else if` ليسلا ضروريّين. لإنشاء شرط، فقط `if` هو الضروري (هذا منطقي، و إلّا فكيف سيكون هناك شرط !).

لا حظ أنّه يمكننا أن نستعمل الكمّ الّذي نريده من الـ `else if`.

عدّة شروط في مرّة واحدة

قد يكون من المهم القيام بعدّة اختبارات في شرط واحد. مثلا، قد تريد اختبار ما إن كان العمر أكبر من 18 و العمر أصغر من 25.
لهذا علينا بتعلم رموز جديدة :

الرمز	المعنى
&&	و
||	أو
!	لا (عكس الشرط)

الاختبار بالواو (and)

لنختبر إن كان العمر في نفس الوقت أكبر من 18 و أقل من 25 يجب كتابة :

```c
1  if (age > 18 && age < 25)
```

`&&` يعنيان "و". بالعربية هذا يعني : "إذا كان العمر أكبر من 18 و إذا كان العمر أصغر من 25".

الاختبار بالأو (or)

لاستخدام "أو"، يجب كتابة الرمزين `||`. لكتابة الرمز `|`، نضغط في لوحة المفاتيح على `Alt Gr` + `6` (باعتبار أن تخطيط لوحة المفاتيح AZERTY فرنسي).

فلنعتبر برنامجا بسيطا، يختبر ما إن كان الشخص قادراً على فتح حساب بنكي أو لا. فلكي يكون قادراً على ذلك يجب أن لا يكون شابّا كثيرا (فلنقل مثلا، ليس تحت 30 سنة) أو لديه الكثير من المال :

```
if (age > 30 || argent > 100000)
{
        printf("Welcome to PicsouBank !");
}
else
{
        printf("Get out of my sight, miserable !");
}
```

الاختبار سيكون ناجحا فقط إذا كان الشخص بعمر أكبر من 30 سنة، أو على الأقل يملك مبلغاً أكبر من 100000 دينار مثلاً.

الإختبار بلا (not)

الرمز الموافق لهذا الاختبار هو علامة التعجّب (!). في علوم الحاسوب، علامة التعجّب تعني "لا". يجب وضع هذه العلامة من أجل عكس الشرط لقول "إن لم يكن هذا صحيحا" :

```
if (!(age < 18))
```

يمكن أن نترجم هذا إلى "إن كان الشخص غير قاصر". لو حذفنا `!` فسيعكس المعنى : "إن كان الشخص قاصرا".

بعض الأخطاء التي يرتكبها المبتدؤون

لا تنس الإشارتين ==

مثلما قلت سابقا، لكي نختبر ما إن كان العمر يساوي 18 نكتب :

```
if (age == 18)
{
        printf("You have just become major !");
}
```

لا تنس وضع علامتي "يساوي" داخل `if`، هكذا : `==`.

إن وضعت علامة `=` واحدة فإن المتغير `age` سيأخذ القيمة 18 (مثلما تعلمنا ذلك سابقا في فصل المتغيرات). نحن نريد أن نختبر قيمة المتغير و ليس تغييرها فاحذر ! الكثير يقع في هذا الخطأ و بالتأكيد فبرامجهم لن تعمل بالشكل المطلوب !

الفاصلة المنقوطة الزائدة

بعض المبتدئين يقومون بإضافة فاصلة منقوطة في نهاية سطر الـ`if` ، و لكن الـ`if` هو شرط و لا نضع فاصلة منقوطة إلّا في نهاية تعليمة. الشفرة التالية لن تعمل كما هو متوقّع لأنّه يوجد `;` في نهاية الشرط.

```
if (age == 18); // Note the semicolon that mustn't be here
{
        printf("You are just major !");
}
```

2.6 المتغيرات المنطقية (Booleans)، أساس الشروط

سندخل الآن في تفاصيل عمل شرط من نوع `if... else`.
في الشروط نعتمد كثيراً على نوع آخر من أنواع البيانات نسميه الـ**boolean**.

بعض الأمثلة البسيطة للفهم

سنبدأ ببعض التجارب قبل أن نقدّم هذا المفهوم. إليك شفرة مصدريّة بسيطة جدّاً أقترح عليك تجريبها :

```
if (1);
{
        printf("It is true !");
}
else
{
        printf("It is false !");
}
```

النتيجة :

```
It is true !
```

؟ لكن، لم نضع شرطا داخل `if` ، هذا عدد فقط. مالّذي يعنيه ؟ لا معنى لهذا.

بلى، ستفهم. استبدل الواحد بالصفر :

```
if (0);
{
        printf("It is true !");
}
else
{
        printf("It is false !");
}
```

النتيجة :

```
It is false !
```

حاول الآن استبدال الصفر بأي عدد صحيح، مثل 4، 15، 226، −10، −36، إلخ. النتيجة دائما : "It is true !".

ملخّص اختباراتنا : إذا وضعنا 0، الاختبار سيعتبر خاطئا، أمّا إن وضعنا 1 أو أيّ عدد آخر، فالاختبار سيكون صحيحا.

الشرح واجب

في الواقع أنه في كلّ مرة تقوم فيها باختبار شرط، فإن الشرط سيقوم بارجاع القيمة 1 إن كان صحيحاً و القيمة 0 إن كان خاطئاً.

مثلا بالنسبة لهذا الشرط الّذي هو `age >= 18` :

```
1 if (age >= 18)
```

لنفرض أن `age` كان 23. إذن، الشرط صحيح، و الحاسوب "سيستبدل" بطريقة ما `age >= 18` بـ1.
بعد ذلك، سيحصل الحاسوب (في رأسه) على `if (1)` . عندما يكون 1، كما رأينا، فسيعتبره صحيحا.

بالمثل، إذا كان الشرط خاطئا، فسيستبدل `age >= 18` بـ0، فالشرط خاطئ، و الحاسوب سيقرأ تعليمات `else` .

فلنجرب مع متغير

فلنجرّب الآن شيئا آخر : وضع نتيجة الشرط في متغير، إن هذا الأمر ممكن مادام الشرط معتبراً من طرف الحاسوب كتعليمة.

```
1 int age = 20;
2 int major = 0;
3
4 major = age >= 18;
5 printf("Major equals : %d\n", major);
```

كما تلاحظ فإن الشرط `age >= 18` أعطى 1 و بالتالي فإن المتغير `major` أخذ القيمة 1 يعني صحيح. يمكنك التأكد بالـ `printf` .

قم بنفس الاختبار مع وضع `age == 10` . هذه المرّة، `major` سيكون 0.

المتغير major متغير منطقي

تذكّر هذا جيّدا : نقول عن متغيّر نجعله يأخذ القيم 1 و 0 أنّه **متغيّر منطقيّ** (boolean).

و هذا كما يلي :

- 0 = خطأ،
- 1 = صحيح.

في الواقع 0 يمثّل خطأ، و كلّ الأعداد الأخرى تعني صحيح (كما رأينا سابقا). و لكن من أجل تبسيط الأمور، لن نستخدم سوى 0 و 1 لقول إن كان الشرط صحيحا أو خاطئا.

في لغة C لا يوجد نوع `boolean`.
على الرغم من ذلك، يمكننا تغطية ذلك باستعمال النوع `int`.

المتغيرات المنطقية في الشروط

عادة، نقوم باختبار `if` على متغيّر منطقيّ :

```
int major = 1;
if (major)
{
        printf("You are major !");
}
else {
        printf("You are minor !");
}
```

بما أن المتغير `major` يساوي 1 فإن الشرط محقق و بالتالي سيظهر على الشاشة : "You are major !".

و هذا عمليّ جدّا، فالشرط أصبح مفهوما بشكل أفضل، فنقرأ `if (major)`، و الّذي يعني "إن كنت بالغاً". الشروط على المتغيّرات المنطقيّة هي إذن سهلة للقراءة و الفهم، ما دمت قد أعطيت أسماء واضحة لمتغيّراتك كما طلبت منك من البداية.

يمكننا أيضا أن نجد شفرة كالتالي :

```
if (major && boy)
```

يعني "إن كان الشخص راشدا و ذكراً". في هذه الحالة، المتغير `boy` أيضا هو متغير منطقي، يساوي 1 إذا كان الشخص ولداً و 0 إذا كان بنتاً. أعتقد أنك فهمت المقصود.

باختصار، المتغيرات المنطقية تسمح لنا بمعرفة ما إن كان الاختبار صحيحاً أم خاطئاً.
هذا مهمّ جدّا و ما شرحته لك سيمكّنك من فهم كثير من الأمور الّتي ستأتي لاحقا.

سؤال صغير : إن كتبنا : `if (major == 1)` فسيعمل، أليس كذلك ؟

هذا صحيح، لكن مبدأ المتغيّرات هي أن نستطيع اختصار عبارة الشرط و جعلها أكثر قابليّة للقراءة. اعترف أنّ `if (major)` تُفهم أحسن، أليس كذلك ؟

تذكّر إذن : إذا كان متغيّرك يحمل عددا (مثل العمر)، قم باختبار من الشكل `if (variable == 1)`.
بالمقابل،إذا كان المتغيّر منطقياً، (أي إمّا 0 أو 1 لقول صحيح أو خطأ)، قم باختبار من الشكل `if (variable)`.

3.6 الشرط switch

الشرط `if ... else` الّذي رأيناه، هو أكثر أنواع الشروط استخداما.
في الواقع، لا يوجد 36 طريقة لكتابة شرط في الـC. الـ `if ... else` يُمكّن من التعامل مع كلّ الحالات.

مع ذلك، `if ... else` يبدو قليلا ... تكراريّا. فلنرَ هذا المثال :

```
1  if (age == 2)
2  {
3          printf("Hello baby !");
4  }
5  else if (age == 6)
6  {
7          printf("Hello kid !");
8  }
9  else if (age == 12)
10 {
11         printf("Hello young !");
12 }
13 else if (age == 16)
14 {
15         printf("Hello teenager !");
16 }
17 else if (age == 18)
18 {
19         printf("Hello adult !");
20 }
21 else if (age == 68)
22 {
23         printf("Hello grandpa !");
24 }
25 else
26 {
27         printf("I have no words ready for your age");
28 }
```

بناء switch

المبرمجون يكرهون الأشياء التكراريّة، لقد رأينا ذلك من قبل.

إذن، من أجل تجنّب التكرار في اختبار قيمة متغيّر وحيد، فقد اخترعوا تعليمة مثل `if ... else`. هذه التعليمة الخاصّة تدعى `switch`. إليكم مثالا نقوم فيه باستعمال `switch`. على المثال السابق :

```
switch (age)
{
        case 2 :
        printf("Hello baby !");
        break;

        case 6 :
        printf("Hello kid !");
        break;

        case 12 :
        printf("Hello young !");
        break;

        case 16 :
        printf("Hello teenager !");
        break;

        case 18 :
        printf("Hello adult !");
        break;

        case 68 :
        printf("Hello grandpa !");
        break;

        default :
        printf("I have no words ready for your age");
}
```

استلهم من مثالي لإنشاء `switch` الخاصة بك. نستخدمها نادرا، لكنّها عمليّة كثيرا لأنّها تجعلنا نكتب قدرا أقل (قليلا) من الشفرة.

الفكرة هي أن نكتب `switch (myVariable)` لكي نقول : سنقوم باختبار حول قيمة المتغير `myVariable`. ثم نقوم بفتح حاضنتين نغلقهما في الأسفل.

بعد ذلك، في داخل الحاضنتين، نتعامل مع كل الحالات : `case 2`، `case 4`، `case 5`، `case 45`، ...

يجب عليك في نهاية كل حالة، أن تضع التعليمة `break;`. إن لم تفعل فإن الحاسوب سينتقل لقراءة التعليمات الموالية التي هي من المفروض محجوزة للحالات الأخرى ! أي أن التعليمة `break;` تجبر الحاسوب على الخروج من الحاضنتين.

في النهاية، `default` هي مثابة الـ `else` الذي تعرفه جيداً الآن. أي أنه إن لم يساوي المتغير أي من الحالات المذكورة فإن الحاسوب يقوم بتشغيل الحالة `default`.

التحكم بقائمة بواسطة الـswitch

الـ `switch` يُستخدم بكثرة لإنشاء قوائم في الكونسول.
أعتقد أنّه الوقت المناسب لبعض التطبيق !

إلى العمل !

نريد أن نظهر في الكونسول قائمة للمستعمل، نستخدم `printf` لعرض مختلف الخيارات المتوفّرة. كل اختيار يرافقه رقم، و على المستعمل أن يدخل رقم الاختيار الذي يريد.
هذا على سبيل المثال ما يجب أن يظهر :

```
=== Menu ===
1. Royal Cheese
2. Mc Deluxe
3. Mc Bacon
4. Big Mac
Your choice ?
```

مهمّتك (إن قبلتها) : ستقوم بإظهار القائمة بالإستعانة بـ `printf` و ستستعمل `scanf` لاسترجاع اختيار المستعمل في متغيّر `choice` و من ثمّ تستعمل `switch` لتختبر الإختيار الذي قام به المستعمل. و في النهاية حسب الحالة ستقول للمستعمل ماذا اختار، هل اختار "Big Mac" أو "Mc Bacon" مثلا.

هيّاء إلى العمل !

تصحيح

هذا هو التصحيح (أتمنّى أنّك قد وجدته بنفسك !) :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char *argv[])
4  {
5          int choiceMenu;
6
7          printf("=== Menu ===\n\n");
8          printf("1. Royal Cheese\n");
9          printf("2. Mc Deluxe\n");
10         printf("3. Mc Bacon\n");
11         printf("4. Big Mac\n");
```

```
12        printf("\nYour choice ? ");
13        scanf("%d", &choiceMenu);
14
15        printf("\n");
16
17        switch (choiceMenu)
18        {
19                case 1:
20                printf("You have chosen the Royal Cheese. Good choice !");
21                break;
22                case 2:
23                printf("You have chosen the Mc Deluxe. Berk, too much sauce..."
                      );
24                break;
25                case 3:
26                printf("You have chosen the Mc Bacon. Well, it goes ;o)");
27                break;
28                case 4:
29                printf("You have chosen the Big Mac. You must be very hungry!")
                      ;
30                break;
31                default:
32                printf("You haven't specified a correct number. You shall not
                      eat anything!");
33                break;
34        }
35
36        printf("\n\n");
37
38        return 0;
39 }
```

و هذا هو العمل !

أتمنى أنك لم تنس الـ `default` في نهاية الـ `switch` ! في الواقع، عندما تبرمج يجب عليك التفكير في كلّ الحالات، فحتّى لو طلبت عددا بين 1 و 4، فسيأتيك دائما أحمق ليكتب 10 أو حتّى "مرحبا"، و هذا ليس ما كنت تنتظره.

باختصار، عليك دائما أن تكون حذرا و لا تثق في المستخدم، لأنّه يمكنه إدخال أيّ شيء. يجب عليك اختبار حالة `default` أيضاً أو `else` في الشروط التي تنشئها بالـ `if`.

> أنصحك أن تألف عمل القوائم في الكونسول، لأنّنا سننشئ عادة برامج فيها و ستحتاجها حتما.

4.6 الثلاثيات : الشروط المختصرة

توجد طريقة أخرى لإنشاء الشروط، أكثر ندرة.

هذا ما يعرف بـ**العبارات الثلاثيّة**.
في الواقع، هي تشبه `if ... else`، باستثناء أنها تكتب على سطر واحد !

و لأنّ إعطاء مثال واحد أحسن من كتابة خطاب طويل، فسوف أعطيك نفس الشرط مرّتين : الأولى باستخدام `if ... else`، و الثانية مطابقة لها باستخدام عبارة ثلاثيّة.

شرط `if ... else` معروف

فلنفرض أنّ لدينا متغيّرا منطقيّا `major` يأخذ القيمة 1 إن كان الشخص راشداً و 0 إن كان قاصراً.
نريد تغيير قيمة المتغيّر `age` حسب المتغيّر المنطقي، نضع فيها 18 إن كان راشدا و 17 إن كان قاصرا. أوافقك الرأي على أنّه مثال غبيّ، و لكنّه يسمح لنا بفهم آليّة عمل العبارات الثلاثيّة.

هكذا يكون باستعمال الـ`if ... else` :

```
if(major)
        age = 18;
else
        age = 17;
```

م تلاحظ أنني قمت بنزع الحاضنتين لأنه لا توجد سوى تعليمة واحدة داخل `if` و أخرى داخل `else`، كما قد شرحت لك سابقا.

نفس الشرط بالثلاثيّة

هذه الشفرة تقوم تماما بنفس عمل الشفرة السابقة، لكنّها مكتوبة بالشكل الثلاثي :

```
age = (major)? 18 : 17;
```

الثلاثيات تسمح، بسطر واحد، بتغيير قيمة متغيّر حسب شرط معيّن. هنا، الشرط هو `major` ببساطة، و لكن كان بالإمكان وضع أي شرط مهما كان طوله. تريد مثالا آخرا ؟
`authorization = (age >= 18) ? 1 : 0;`.

علامة الإستفهام تمكّن من قول "هل هو راشد ؟". إن كان الأمر كذلك فنضع القيمة 18 في `age`، و إلّا (النقطتان تعنيان `else` هنا) نضع القيمة 17.

الثلاثيّات ليست ضروريّة جدّا، شخصيّا، لا أستخدمها إلّا قليلا لأنّها تجعل قراءة الشفرة أكثر صعوبة نوعا ما. لكن يجب عليك دراستها تحسّبا ليوم تقع فيه على شفرة مليئة بالثلاثيّات !

ملخّص

- **الشروط** أمور قاعدية في البرمجة، و باستخدامها **نتخّذ قرارات** على حسب قيمة متغيّر ما.
- الكلمات المفتاحية `if` ، `else` ، `else if` تعني -على التوالي- إذا، و إلا ، و إلا فإذا. يمكننا استعمال `else if` بالقدر الذي نريد.
- **المتغير المنطقي** هو متغير يشير إذا ما كان الشيء صحيحا أو خاطئاً، الصفر يعني خطأ و الواحد يعني صحيح (أي قيمة مختلفة عن 0 تعتبر صحيح). نستخدم النوع `int` للتصريح عن هذه المتغيّرات لأنها ليست سوى أعداد في الواقع.
- الـ`switch` هي بديل للـ`if`، تستخدم عندما نريد دراسة حالات قيمة متغيّر ما. يسمح بجعل الشفرة أكثر وضوحاً إذا كنت تريد اختبار عدد معتبر من الحالات. إذا كنت تستخدم كثيرا من `else if` فهذه عادة ما تكون إشارة إلى أن `switch` تكون أحسن في جعل الشفرة أسهل للقراءة.
- **الثلاثيات** هي شروط مختصرة جدّا تسمح بإعطاء قيمة لمتغيّر حسب نتيجة اختبار. نستخدمها بشكل قليل لأنها قد تجعل الشفرة أقل وضوحاً.

الفصل 7

الحلقات التكرارية (Loops)

بعدما تعلمنا كيف ننشئ شروطا بلغة C، سنكتشف معاً **الحلقات التكراريّة** (Loops). ما هي الحلقة ؟ هي تقنية تسمح بتكرار نفس التعليمات عدة مرات. و ستساعدنا كثيرا من الآن و صاعدا خاصة في العمل التطبيقي الأوّل الذي ينتظرنا بعد هذا الفصل.

استرخ : هذا الفصل سيكون سهلاً. لقد تعرفنا سابقا على ما تعنيه المتغيّرات المنطقية (booleans) و الشروط (conditions) في الفصل السابق، و بذلك كنا قد تخلصنا من عمل كبير. من الآن فصاعداً ستكون الأمور سلسة أكثر و لن يكون في العمل التطبيقي القادم الكثير من المشاكل.

فلننتهز الفرصة، لأننا لن نتأخر في الدخول في الجزء الثاني من الكتاب. سيكون من الجيّد لك أن تنتبه !

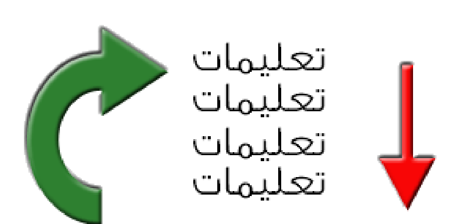
1.7 ماهي الحلقة ؟

كما قلت سابقاً : هي عبارة عن تعليمة تسمح لنا بتكرار نفس التعليمات عدة مرات.

تماما مثل الشروط، توجد طرق عديدة لإنشاء الحلقات. و لكن مهما اختلفت الطرائق فالهدف واحد : تكرار تعليمات لعدد معيّن من المرات.
لدينا في لغة C ثلاثة أنواع من الحلقات :

- `while`
- `do ... while`
- `for`

في جميع الحالات يبقى المخطط نفسه :



و هذا ما سيحصل بالترتيب :

1. الجهاز يقرأ التعليمات من الأعلى إلى الأسفل كالعادة.
2. ما إن يصل لنهاية الحلقة يتوجه نحو التعليمة الأولى.
3. يعيد بعدها قراءة التعليمات كلها من الأعلى إلى الأسفل.
4. يصل لنهاية الحلقة و يعاود الرجوع للأول من جديد و هكذا ...

المشكلة في هذا النظام هو أننا إن لم نقم بإيقافه، فالجهاز قادر على تكرار نفس التعليمات إلى مالانهاية ! و لن يتذمّر، أنت تعرف : هو يفعل ما تأمره أنت بفعله ... يمكنه أن يعلق في حلقة غير منتهية، و هذا النوع من الحالات يعتبر مصدر خوف بالنسبة للمبرمجين.

و هنا نجد ... الشروط ! فعندما ننشئ حلقة نقوم دائما بتعريف شرطها. هذا الشرط يعني "كرّر الحلقة دون توقف مادام هذا الشرط صحيحا".

كما قلت، فهناك عدة طرق للقيام بذلك و سنبدأ من دون تأخير بإنشاء حلقة من نوع `while` في الـC.

2.7 الحلقة while

هكذا نشكل حلقة `while` :

```
1 while (/* Condition */)
2 {
3         // The instructions that we want to repeat
4 }
```

لا يوجد أبسط من هذا. الكلمة `while` تعني "مادام" ، لذا نقول للجهاز: مادام الشرط صحيحا، كرر التعليمات المتواجدة بين الحاضنتين.

أقترح عليك أن نقوم باختبار بسيط : سنطلب من المستعمل ادخال العدد 47، مادام لم يقم بإدخاله، نطلب منه إعادة إدخاله مجدداً ... و لن يتوقف البرنامج حتى يقوم المستعمل بإدخال العدد 47 (نعم أعرف، إنه عمل شيطاني) :

```
1 int entredNumber = 0;
2 while (entredNumber != 47)
3 {
4         printf("Enter the number 47 ! ");
5         scanf("%d", &entredNumber);
6 }
```

أنظر إلى الاختبار الذي قمت به، للعلم أنني تعمدت الخطأ ثلاث مرات :

```
Enter the number 47 ! 10
Enter the number 47 ! 27
Enter the number 47 ! 40
Enter the number 47 ! 47
```

يتوقف البرنامج بعد إدخال العدد 47.
هذه الحلقة `while` ستتكرر مادام المستعمل لم يدخل العدد 47، لا يوجد أسهل من هذا.

الآن لنجعل الأمر ممتعاً أكثر : نريد من الحلقة أن تتوقف بعد عدد معين من التكرارات.
لهذا سنستعين بمتغير `counter` الذي سيأخذ القيمة 0 في بداية البرنامج ثم نقوم **بزيادته**، هل تتذكر ما قلناه في الفصل السابق حول الزيادة (incrementation) ؟ التي تنص على إضافة 1 لمتغير حينما نكتب `variable++`.

إقرأ جيدا الشفرة المصدرية التالية و حاول التمعن فيها و فهمها :

```
int counter = 0;
while (counter < 10)
{
        printf("Hello !\n");
        counter++;
}
```

النتيجة :

```
Hello !
Hello !
Hello !
Hello !
Hello !
Hello !
Hello !
Hello !
Hello !
Hello !
```

البرنامج يكرر عشر مرات العبارة "Hello !"

؟ كيف يعمل هذا بالتحديد ؟

1. في البداية لدينا متغير `counter` مهيأ على القيمة الإبتدائية 0.
2. الحلقة `while` تأمر بالتكرار مادامت قيمة المتغير `counter` أصغر من 10. بما أن قيمة المتغير `counter` هي 0 في البداية، فإننا ندخل في الحلقة لأن الشرط محقق.
3. نقوم بإظهار الرسالة "Hello !" على الشاشة باستخدام الدالة `printf`.
4. نقوم بزيادة قيمة المتغير `counter` بفضل التعليمة `counter++;`. كان المتغير `counter` يحمل القيمة 0، أمّا الآن فهو يحمل القيمة 1.
5. نصل لنهاية الحلقة (حاضنة الإغلاق) : نعيد العملية من جديد و نتأكد ما إن كان الشرط محققاً أي ما إن كانت قيمة المتغير أصغر من 10 ؟ في هذه الحالة نعم لأن المتغير `counter` يحمل القيمة 1 و هي أصغر من 10 إذا سنمرّ بنفس التعليمة داخل الحاضنتين.

و هكذا دواليك، `counter` تصبح 1، 2، 3، ... ، 8، 9 ثمّ 10. حينها يصبح الشرط `counter < 10` غير محقق، و عندها نخرج من الحلقة.

و يمكننا ملاحظة أن قيمة المتغير `counter` تزيد في كلّ مرة بواحد. يمكننا التأكّد بـ `printf` :

```
int counter = 0;
while (counter < 10)
{
        printf("counter = %d\n", counter);
        counter++;
}
```

```
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
counter = 5
counter = 6
counter = 7
counter = 8
counter = 9
```

إن كنت قد فهمت المثال السابق فقد فهمت كلّ شيء !
يمكنك الاستمتاع بتجربة أعداد أكبر من 10 (مثلا 100 أو أي عدد آخر). كان هذا سيفيدني كثيرا في صغري لكتابة العقوبات الّتي كان يجب عليّ تكرارها مائة مرّة.

احذر من الحلقات غير المنتهية !

عندما تنشئ حلقة فتأكّد دائما من جعلها **قادرة على التوقف في لحظة معينة !** إن كان الشرط محققاً دائماً، فلن يتوقف البرنامج أبداً ! و هذا مثال على حلقة غير منتهية :

```
while (1)
{
        printf("Infinite loop\n");
}
```

تذكر ما قلناه بخصوص القيم المنطقية (booleans) : فصحيح = 1 و خاطئ = 0. هنا، الشرط محقق دائماً و بهذا فإن البرنامج سيستمرّ في كتابة العبارة "Infinite loop" بدون توقّف !

> م
> لإيقاف برنامج كهذا في الويندوز، ليس هناك حلّ سوى الضغط على الزر `x` الملوّن بالأحمر في أعلى النافذة بينما على مستخدمي اللينكس الضغط على `Ctrl` + `C` لخروج من البرنامج.

لكن توخ الحذر. تجنب دائما الوقوع في الحلقات غير المنتهية، بالرغم من أنها قد تكون مفيدة في بعض الحالات، خصوصا في برمجة ألعاب الفيديو، هذا ما سنراه لاحقاً.

3.7 الحلقة do ... while

هذا النوع من الحلقات يشبه كثيرا `while` إلا أنه قليل الاستعمال عادة.
الشيء الوحيد الذي يتغيّر هو مكان الشرط في الحلقة. فعِوَض أن يكون الشرط في بداية الحلقة، فهو موجود في نهايتها :

```
int counter = 0;
do
{
        printf("Hello !\n");
        counter++;
} while (counter < 10);
```

ما الذي تغير ؟
هذا أمر بسيط فالحلقة `while` يمكن أن لا يتم تشغيلها أبداً إذا كان الشرط غير محقق منذ البداية. فمثلا لو وضعنا القيمة 50 كقيمة ابتدائية للمتغير `counter` فإن الشرط لن يكون محققاً و لذلك لا ندخل في الحلقة أبدا.
بالنسبة للحلقة `do ... while` فالأمر مختلف : هذه الحلقة تتكرر على الأقل مرة واحدة. في الواقع، اختبار الشرط يتم في نهاية الحلقة كما يمكنك الملاحظة. فلو أعطينا القيمة الإبتدائية 50 للمتغير `counter` فإن التعليمة بين الحاضنتين سيتم تشغيلها مرة واحدة.

لهذا يجب أحيانا استخدام الحلقات من هذا النوع، لنضمن تكرارها للعملية على الأقل مرة واحدة.

> ! هناك استثناء في الحلقة `do ... while` يغفل عنه الكثير من المبتدئين، و هو وجود فاصلة منقوطة في نهاية الحلقة ! لا تنس وضع واحدة بعد الـ`while` و إلا ستظهر لك أخطاء أثناء الترجمة !

4.7 الحلقة for

نظرياً، الحلقة `while` تلبي رغبتنا في أي نوع كان من التكرارات.
سابقا، كما رأينا في الشرط `switch` فإنه توجد أيضا طريقة أخرى لكتابة حلقة بشكل مختصر أكثر.

الحلقات `for` مستخدمة كثيرا في البرمجة. لا أملك إحصائيات لكن كن واثقاً أنك ستستخدم الحلقة `for` أكثر من الحلقة `while`. و لهذا يجب عليك أن تجيد استخدام كلتا الحلقتين.

و كما قلت لك فالحلقة `for` ماهي إلا عبارة عن طريقة أخرى لكتابة الحلقة `while` التي رأيناها قبل قليل.
من المثال السابق لدينا :

```
int counter = 0;
while (counter < 10)
{
        printf("Hello !\n");
        counter++;
}
```

الشفرة المصدرية المكافئة باستعمال الحلقة `for` :

```
int counter;
for (counter = 0; counter < 10; counter++)
{
        printf("Hello !\n");
}
```

ما الاختلاف ؟

- تلاحظ أننا لم نهيّئ المتغير `counter` على قيمة ابتدائية تساوي 0 عند إنشائه (لكنّه كان بإمكاننا فعل ذلك).
- هناك الكثير من التفاصيل داخل القوسين بعد `for` ، سنفصل ذلك لاحقاً.
- لا توجد التعليمة `counter++;` في الحلقة.

فلنهتم بما يوجد بين القوسين فهو كل ما يهم في الحلقة `for` . هناك ثلاث تعليمات مختلفة نفصل فيما بينها بفواصل منقوطة.

- الأولى هي **التهيئة** (initialisation) : هذه التعليمة تعمل على تحضير المتغير `counter` . في حالتنا، تهيئته بالقيمة 0.
- الثانية هي **الشرط** (condition) : كما رأينا في الحلقة `while` فإن الشرط يخبرنا ما إن كان يجب تكرار الحلقة أم لا. مادام الشرط محققاً فإن الحلقة تستمر في تكرار التعليمات.
- أخيرا، **الزيادة** (incrementation) : هذه التعليمة يتم تنفيذها بعد كل التعليمات أي في نهاية الدورة الواحدة و هي تقوم بتعديل قيمة المتغير `counter` في كلّ مرة. في غالب الأحيان نقوم بالزيادة، لكن يمكننا أن نستعمل الإنقاص (مثلا `counter--`)، كما يمكننا القيام بعمليّات أخرى، مثلا زيادة قيمة العداد بـ2 (`counter+=2`).

في النهاية، نلاحظ أن حلقة `for` ماهي سوى حلقة كتابة مختصرة. تعلّم كيفية استعمالها لأنك ستحتاجها بكثرة !

ملخّص

- الحلقات هي تعليمات تسمح لنا بتكرار سلسلة من التعليمات عدة مرات.
- توجد كثير من الحلقات : `while` ، `for` و `do ... while` . كلّ منها يُستعمل في الحالة الّتي يكون فيها مناسبا أكثر.
- الحلقة `for` هي التي سنستعملها بكثرة في التطبيقات. نقوم فيها غالباً بزيادات أو إنقاصات للمتغيرات.

الفصل 8

عمل تطبيقي : "أكثر أو أقل"، لعبتك الأولى

نصل اليوم إلى أول عمل تطبيقي. الهدف هو أن أريك أنك قادر على برمجة الكثير من الأشياء بما علّمتك إياه. لأنه في الواقع، الجانب النظري للغة أمر جيّد لكننا إن كنّا لا نعرف كيف نطبّق ما تعلّمناه بشكل سلس فلا داعي لإهدار وقتنا بتعلم المزيد.

صدّق أو لا تصدّق، يسمح لك مستواك الآن ببرمجة أول برنامج ممتع. إنّه لعبة كونسول (أذكّرك بأننا سنصل للبرامج بنافذة لاحقا). مبدأ عمل اللعبة سهل للغاية، وسهل البرمجة، و لهذا اخترتها لتكون موضوع أول عمل تطبيقي لك.

1.8 تجهيزات و نصائح

مبدأ عمل البرنامج

قبل كلّ شيء، سأشرح عمل برنامجنا. إنّه لعبة صغيرة نسمّيها "أكثر أو أقل".

مبدأ عمل اللعبة هو التالي.

1. الجهاز يختار عشوائيا عددا من 1 إلى 100.
2. يطلب منك أن تخمن عددا و بالتالي ستختار بدورك عددا من 1 إلى 100.
3. يقوم الجهاز بمقارنة العدد الذي كتبته بالعدد "الغامض" الّذي حصل عليه عشوائيّا، ثم يقول لك ما إن كان العدد الغامض أصغر أو أكبر من العدد الذي اخترته أنت.
4. ثم يقوم الجهاز بإعادة طلب العدد منك.
5. ثم يقول لك ما إن كان العدد الغامض أصغر أو أكبر من العدد الذي اخترته أنت.
6. و هكذا تستمر العملية حتى تجد أنت ذلك العدد.

و الهدف من اللعبة هو أن تجد العدد الغامض في أقل عدد ممكن من المحاولات بالطبع.

و هذه "لقطة شاشة" مما يجب أن تكون عليه اللعبة في طور التنفيذ :

```
What's the number ? 50
Greater !
What's the number ? 75
Greater  !
What's the number ? 85
Lesser !
What's the number ? 80
Lesser !
What's the number ? 78
Greater !
What's the number ? 79
Bravo, you have found the mysterious number !!!
```

اختيار عدد عشوائي

> ؟ لكن كيف يختار الجهاز عدداً عشوائياً ؟ أنا لا أجيد فعل هذا !

صحيح، نحن لا نجيد كيفية توليد عدد عشوائي. و يجب القول أن طلب ذلك من الحاسوب ليس أمراً سهلاً : هو يجيد القيام بعمليات حسابية، لكن أن يستخرج عدداً عشوائيا، هذا أمر لا يجيد فعله !

في الواقع، لـ"محاولة" الحصول على عدد عشوائيّ، يجب القيام بحسابات معقّدة للحاسوب، و هذا ما يعود في النهاية إلى القيام بحسابات !

لكي نفعل هذا، نميز حلين.

- إما أن نطلب من المستعمل أن يقوم باختيار عدد عشوائي في بداية اللعبة بواسطة الدالة `scanf`. هذا يستلزم لاعبين : واحد يقوم بإدخال العدد الغامض و الآخر يحاول تخمينه بعد ذلك.
- نجرّب طريقة ثانية لجعل الجهاز يختار وحده العدد. الشيء الجيّد هنا هي أنّك ستتمكن من لعب اللعبة لوحدك. أمّا الشيء السيّء هو ... أنّني مضطرّ لأن أشرح لك كيف تقوم بذلك !

طبعا سنقوم بالاختيار الثاني و لكن إن أردت تجريب الحل الأول فيما بعد فلا شيء يمنعك.

لاختيار عدد عشوائي نستعمل الدالة `rand`. و هي دالة تسمح باختيار عدد بطريقة عشوائية. لكنّنا نريد عدداً بين 1 و 100 مثلا (لأننا إن لم نعرف الحدود فستصبح اللعبة معقّدة جدّا).

للقيام بذلك، سنستخدم العبارة التالية (لا يمكنني أن أطلب منك تخمينها هي أيضاً !) :

```c
srand(time(NULL));
mysteriousNumber = (rand() % (MAX — MIN + 1)) + MIN;
```

السطر الأول (الذي به `srand`) يسمح بتهيئة مولّد الأرقام العشوائية. نعم الأمر صعب قليلاً كما أخبرتك.
`mysteriousNumber` هو متغير سيحمل العدد العشوائي المختار.

> !
> التعليمة `srand` لا يجب أن تشغّل إلا مرة واحدة (في بداية البرنامج). فيجب وضع `srand` مرة واحدة في البداية، و مرة واحدة فقط.
> بعدها يمكنك استعمال القدر الذي تريده من الدالة `rand` لاختيار العدد العشوائي. و لكن يجب ألّا يقرأ جهازك التعليمة `srand` مرتين، لا تنس ذلك.

`MAX` و `MIN` هما ثابتان، الأول هو العدد الأقصى (100) و الثاني هو العدد الأدنى (1)، و أنصحك بتعريف الثابتين في بداية البرنامج هكذا :

```
1 const int MAX = 100, MIN = 1;
```

تضمين المكتبات اللازمة

كي يشتغل برنامجك بشكل صحيح، يجب أن تقوم بتضمين المكتبات `stdlib` و `stdio` و `time` (الأخيرة تستعمل من أجل الأعداد العشوائية).
يجب إذن أن يبدأ برنامجك بالتالي :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
```

يكفي شرحا !

حسنا، سأتوقف هنا لأنه لو أكملت سأعطيك الشفرة الخاصة ببرمجة اللعبة كاملة !

> م
> لتوليد الأعداد العشوائية، اضطررت لإعطائك شفرة "جاهزة" دون أن أشرح كيف تعمل بالضبط. عادة، لا أحبّ فعل هذا، لكن لا يوجد خيار هنا لأنني لا أريد تعقيد الأشياء كثيرا حاليّا.
> تأكّد أنه ستتعلم فيما يلي الكثير من المفاهيم التي ستسمح لك بفهم هذه الأشياء بنفسك.

باختصار، أنت تعرف الكثير. لقد شرحت لك المبدأ و أعطيتك صورة عن البرنامج لحظة التشغيل.
بعد كل هذا أنت قادر على كتابة البرنامج لوحدك.

حان وقت العمل ! بالتوفيق !

2.8 التصحيح !

توقف ! من الآن سأجمع الأوراق.

سأعطيك طريقتي الخاصة لحلّ التطبيق. بالطبع هناك طرائق جيّدة عديدة لفعل ذلك. إن كانت لديك شفرة مصدريّة تختلف عن شفرتي و كانت تشتغل، فمن المحتمل أن تكون جيّدة أيضاً.

تصحيح "أكثر أو أقل"

هذا هو التصحيح الذي أقترحه :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main ( int argc, char** argv )
{
	int mysteriousNumber = 0, inputNumber = 0;
	const int MAX = 100, MIN = 1;
	// Generation of random number
	srand(time(NULL));
	mysteriousNumber = (rand() % (MAX - MIN + 1)) + MIN;

	do
	{
		// We request the number
		printf("What's the number? ");
		scanf("%d", &inputNumber );
		// We compare between inputNumber and mysteriousNumber
		if (mysteriousNumber > inputNumber )
		printf("Greater !\n\n");
		else if (mysteriousNumber < inputNumber )
		printf("Lesser !\n\n");
		else
		printf ("Bravo, you have found the mysterious number !!!\n\n");
	} while (inputNumber != mysteriousNumber);
	return 0;
}
```

الملف التنفيذي و الشفرة المصدرية

لمن يريد ذلك، اللعبة و الشفرة المصدرية متوفّران للتنزيل من هنا :

https://openclassrooms.com/uploads/fr/ftp/mateo21/plusoumoins.zip (7 Ko)

> م
> الملف التنفيذي (`.exe`) مترجم للويندوز، لذا إن كنت تستخدم نظام تشغيل آخر فيجب عليك إعادة ترجمة البرنامج ليشتغل عندك.

يوجد مجلّدان، واحد فيه الملف التنفيذي (مترجم للويندوز، أكرّر) و آخر فيه ملفات الشفرة المصدرية.

في حالة "أكثر أو أقل"، المصادر بسيطة جدّا : يوجد فقط الملف `main.c`.
لا تفتح هذا الملفّ مباشرة، افتح أوّلا البيئة التطويرية الّتي تفضّلها، ثم أنشئ مشروعا جديداً من نوع Console و قم باستيراد الملف `main.c`. يمكنك ترجمة البرنامج من جديد للتجريب كما يمكنك التعديل عليه إن أردت.

الشرح

سأشرح الآن الشفرة المصدريّة الخاصة بي بدءً من الأعلى :

توجيهات المعالج القبلي

هي الأسطر التي تبدأ بعلامة `#` في الأعلى. و هي تقوم بتضمين المكتبات التي نحتاج إليها.
أعطيتها لك جاهزة أعلاه، لذلك فإن استطعت ارتكاب خطأ هنا، فأنت قويّ جدّا !

المتغيرات

لم نحتج الكثير منها.
احتجنا فقط واحدا للعدد الذي يدخله المستعمل (`inputNumber`) و ثانٍ لكي نسمح للجهاز بتوليد عدد عشوائي (`mysteriousNumber`).

قمت بتعريف الثوابت كما ذكرت في بداية هذا الفصل. الشيء الجيد أيضاً هو إمكانية تحديد صعوبة اللعبة (بوضع أعلى قيمة هي 1000 مثلا)، يكفي أن نعدّل على هذا السطر و نعيد ترجمة البرنامج.

الحلقة

لقد اخترت الحلقة `do ... while`. نظريّا، حلقة `while` بسيطة يمكنها أن تقوم بالمهمة أيضا، لكنّي وجدت بأن استخدام `do ... while` منطقيّ أكثر.

لماذا ؟ لأنه إن كنت تتذكر، `do ... while` هي حلقة تسمح لنا بتشغيل التعليمات بين الحاضنتين على الأقل مرّة واحدة. و نحن سنطلب من المستعمل إدخال العدد الغامض لمرة واحدة على الأقل أيضاً (فالمستعمل غير قادر على تخمين العدد في أقل من محاولة واحدة، إلا إن كان شخصاً خارقاً !).

في كلّ دورة من الحلقة نطلب من المستعمل إدخال عدد. نقوم بتخزين القيمة في المتغير `inputNumber`.
بعدها، نقارن المتغير السابق بالمتغير `mysteriousNumber`. هنا نجد ثلاثة احتمالات :

- يكون العدد الغامض أكبر من العدد الذي أدخله المستعمل و بالتالي تظهر على الشاشة العبارة "Greater !".
- يكون العدد الغامض أصغر من العدد الذي أدخله المستعمل و بالتالي تظهر على الشاشة العبارة "Lesser !".
- يكون العدد الغامض لا أكبر و لا أصغر من الّذي أدخله المستعمل، أي أنّه مساوٍ بالضرورة ! و بالتالي تظهر على الشاشة العبارة "Bravo, you have found it !".

يجب أن تضع شرطا للحلقة. و هذا سهل للإيجاد : نعيد تشغيل الحلقة **مادام العدد الذي تم إدخاله لا يساوي العدد الغامض.**
و ما إن يتم إدخال العدد المطلوب تتوقف الحلقة و بالتالي ينتهي البرنامج هنا.

أفكار للتحسين

لم تعتقد أنّي أريد للعمل التطبيقي أن ينتهي هنا ؟ أنا أصرّ على أن تقوم بتحسين البرنامج، من أجل التدريب. و لتتأكد بأنه مع التدريب ستتطور قدراتك ! من يعتقد أنه يقرأ الدروس و لا يطبّق ما جاء فيها فهو مخطئ، أقولها و أكررها !

لقد عرفت أنّه لديّ خيال واسع، و حتى بكونه برنامجاً سهلاً فأنا أملك أفكارا لتطويره !

لكن احذر هذه المرة فلن أعطيك التصحيح، عليك أن تدبّر أمرك بنفسك ! فإن كانت لديك حقّا مشاكل، فيمكنك زيارة منتديات OpenClassrooms من أجل طلب المساعدة من الآخرين.

- **ضع عدادا للمحاولات**، و هو متغير نقوم بزيادة قيمته في كلّ مرة عندما لا يجد المستعمل الرقم الغامض. بمعنى آخر في كل مرة يعود فيها البرنامج لتشغيل الحلقة، و عند انتهاء اللعبة تقوم بإظهار عدد المحاولات للاعب هكذا مثلا : "Bravo ! you have found in 3 tries only !".
- في هذا البرنامج، عندما يجد اللاعب العدد الغامض يتوقف البرنامج، لماذا لا نجعل البرنامج يسأله ما إن أراد **جولة ثانية ؟**
إذا أردت تطبيق الفكرة فيجب عليك وضع حلقة تشمل تقريبا كل برنامجك. و هذه الحلقة تتكرر مادام اللاعب لم يطلب إيقاف البرنامج، و أنصحك بإضافة متغير منطقي اسمه مثلا `continuePlaying` ذي قيمة ابتدائية مساوية لـ1. إذا طلب اللاعب إيقاف البرنامج، تُغيّر القيمة لـ0 و يتوقف البرنامج.
- **أنشئ وضع لاعبين !** يعني بالإضافة إلى وضع اللاعب الواحد، ضع إمكانية اشتراك لاعبين !
لهذا عليك بصنع قائمة (menu) في البداية لتطلب من المستخدم اختيار لاعب واحد أو لاعبين إثنين. الفرق بين الوضعين هو توليد العدد الغامض، ففي الحالة الأولى نستعمل الدالة `rand` و في الحالة الثانية نستعمل الدالة `scanf` !
- **إنشاء مستويات مختلفة لصعوبة اللعبة.** يمكنك وضع قائمة في البداية ليختار المستخدم فيه المستوى و هذا مثال :
  - 1 = بين 1 و 100،
  - 2 = بين 1 و 1000،
  - 3 = بين 1 و 10000.

لفعل هذا يجب التعديل على الثابت `MAX`. بطبيعة الحال لا يمكننا أن نسميه ثابتا إن كانت قيمته تتغير أثناء البرنامج ! قم بتغيير اسم هذا المتغير إلى `maximumNumber` (و لا تنس نزع الكلمة المفتاحية `const` لأنه ببقاءها يبقى ثابتا !). تتغير قيمة هذه المتغير حسب المستوى المختار.

ستساعدك هذه التحسينات على الإشتغال قليلاً. لا تتردد في إيجاد أفكار أخرى لتطوير هذه اللعبة، أعلم أنّه يوجد ! و لا تنس أن المنتديات في خدمتك في حالة واجهت صعوبات.

الفصل 9

الدوال (Functions)

ننتهي من الجزء الأول من الكتاب (المبادئ) بهذا المفهوم المهم الذي يتكلّم عن الدوال في لغة C. كل البرامج في لغة C ترتكز على المبدأ الذي سأشرحه في هذا الفصل.

سوف نتعلم هيكلة برامجنا وتقسيمها لعدة أجزاء، تقريباً كما لو كنّا نلعب لعبة Lego.
البرامج الكبيرة في لغة C ماهي إلا تجميعات لأجزاء صغيرة من الشفرات المصدرية، و هذه الأجزاء هي بالضبط ما نسمّيه ... بالدوال !

1.9 إنشاء و استدعاء دالة

رأينا في الفصول السابقة بأن برامج الـC تبدأ بدالة تدعى `main`. لقد أعطيتك مخططاً تلخيصيّاً لتذكيرك ببعض الكلمات المفتاحية :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Hello world!\n");
    return 0;
}
```

توجيهات المعالج القبلي

تعليمات (أوامر)

دالة

في الأعلى، نجد توجيهات المعالج القبلي (اسم معقد سأرجع لشرحه في وقت لاحق). من السهل التعرّف على هذه التوجيهات : هي تبدأ بإشارة `#` و هي غالباً توضع في أعلى الملف المصدري.

لقد قلت لك بأن البرنامج في الـC يبدأ بالدالة `main`. أؤكد لك، هذا صحيح ! إلا أننا في هذه الحالة بقينا داخل الدالة `main` و لم نخرج أبداً منها. أعد قراءة الشفرات المصدرية في الدروس السابقة و سترى : لقد بقينا دائما داخل الحاضنتين الخاصتين بالدالة الرئيسية.

؟ حسناً، هل من السيء فعل ذلك ؟

لا، هذا ليس "سيئا". لكن ذلك خلاف ما يفعله المبرمجون بلغة C حقيقة.
بل و إن كل البرامج تقريباً لا تكون مكتوبة فقط داخل حاضنتي الدالة `main` . لحدّ الآن كانت البرامج التي نكتبها صغيرة و لهذا فهي لم تطرح أي مشكل، لكن تخيّل معي برامج ضخمة تحتوي على آلاف الأسطر من الشفرة المصدرية ! لو كانت كلّ الأسطر مكتوبة داخل حاضنتي الدالة الرئيسية لأصبحنا في السوق.

سنتعلّم الآن كيف ننظّم عملنا. سنبدأ بتقسيم برامجنا إلى قطع صغيرة (تذكّر فكرة Lego الّتي حدّثتك عنها قبل قليل). كلّ قطعة تسمّى **دالّة**.

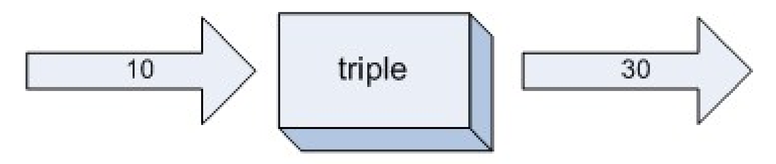
الدالة تقوم بتنفيذ تعليمات و تقوم بإرجاع نتيجة. هي عبارة عن **قطعة من الشفرة المصدرية** تعمل على القيام بمهمة معينة.
نقول بأن الدالة تملك قيمة الإدخال و قيمة الإخراج. المخطط التالي يمثل المبدأ الذي تعمل به الدالة :



حينما نقوم باستدعاء دالة، نمرّ بثلاثة خطوات.

1. **الإدخال** : نقوم بإدخال المعلومات إلى الدالة (بإعطائها معلومات تعمل عليها).
2. **الحسابات** : تقوم الدالة بعمل حسابات على المعلومات التي تم ادخالها.
3. **الإخراج** : بعد أن تنتهي الدالة من الحسابات تعطينا النتيجة على شكل قيمة الإخراج أو الإرجاع.

يمكن أن نتصوّر دالة تسمى `triple` تضرب العدد في 3. بالطبع الدوال في الغالب هي أكثر تعقيداً من هذا.



تضرب الدالة **Triple** العدد الداخل إليها في الرقم ٣

هدف الدوال إذا هو تبسيط الشفرة المصدرية، لكي لا نضطر إلى إعادة كتابة نفس الشفرة المصدرية عدّة مرات على التوالي.

أُحْلُم قليلاً : لاحقاً، سننشئ مثلا دالة اسمها `showWindow` تقوم بفتح نافذة في الشاشة. ما إن نكتب الدالة (المرحلة الأصعب)، لن يتبقّ لنا سوى القول "أيتها الدالة `showWindow` ، أظهري لي النافذة !". يمكننا أيضاً كتابة دالة `moveCharacter` تهدف إلى تحريك شخصية ما في اللعبة، إلخ.

مخطط دالة

لقد تكوّنت لديك فكرة على الطريقة التي تعمل بها الدالة `main`.
و مع ذلك يجب أن أريك كيف نقوم بإنشاء دالة.

الشفرة المصدرية التالية تمثّل دالة تخطيطياً. هذا نموذج للحفظ :

```
type functionName(parameters)
{
        // We write the instructions here
}
```

أنت تعرف شكل الدالة `main`.
إليك ما عليك فهمه بخصوص المخطط.

- `type` (نوع قيمة الإخراج) : هو نوع الدالة. مثل المتغيرات، للدوال أنواعها الخاصة. هذا النوع يعتمد على القيمة التي ترجعها الدالة : إن كانت الدالة ترجع عدداً عشرياً، فسنضع بالتأكيد الكلمة المفتاحية `double`، أما إن كانت ترجع عدداً صحيحاً، سنضع النوع `int` أو `long` مثلا. و لكن يمكن أيضاً إنشاء دوال لا ترجع أي شيء ! هناك إذا نوعان من الدوال :
  - دوال ترجع قيمة : تعطيها أحد الأنواع الّتي نعرفها كـ `int` أو `char` أو `double`، إلخ.
  - دوال لا ترجع أية قيمة : نعطيها نوعا خاصا يدعى `void` (و الذي يعني الفراغ).
  - `functionName` : هو اسم الدالة. يمكنك أن تسمي الدالة مثلما تريد لطالما تحترم القواعد التي تتبعها في تسمية المتغيرات (لا للأحرف التي تحتوي على العلامات الصوتية (accents)، لا فراغات، إلخ).
  - `parameters` : (هي قيم الإدخال) : داخل قوسين، يمكنك أن تبعث معاملات للدالة. هي القيم التي ستعمل بها الدالة.

  > م
  > يمكنك أن تبعث القدر الذي تريد من المعاملات. كما يمكنك ألا تبعث أية معامل، و لكن نادراً ما يُستخدم هذا.

  مثلاً، بالنسبة للدالة `triple`، أنت تبعث عدداً كمعامل. الدالة "تسترجع" العدد و تضربه في 3. تقوم بعد ذلك بإرجاع نتيجة حساباتها.
  - بعد ذلك، نجد **الحاضنتين** اللّتان تشيران إلى بداية الدالة و نهايتها. داخل الحاضنتين، تضع التعليمات التي تريدها. بالنسبة للدالة `triple`، يجب أن تكتب التعليمات التي توافق ضرب قيمة الإدخال في 3.

الدالة إذا هي عبارة عن آلية تتلقّى قيم إدخال (المعاملات) و ترجع قيمة إخراج.

إنشاء دالة

فلنرى مثالا تطبيقيّا دون مزيد من التأخير : الدالة `triple` التي حدثتك عنها منذ قليل. فلنقل أن هذه الدالة تتلقّى عدداً صحيحاً من نوع `int` و أنها تُرجع عدداً صحيحاً أيضاً من نوع `int`. هذه الدالة تضرب العدد الذي نعطيها في 3 :

```
int triple (int number)
{
        int result = 0;
        result = number □ 3; // We multiply the input number by 3
        return result; // We return the result as an output value
}
```

هاهي أول دالة لك ! شيء مهمّ للغاية : كما ترى، الدالة من نوع `int`. فهي مجبرة على أن ترجع قيمة من نوع `int`.

داخل القوسين، نجد المتغيرات التي تتلقّاها الدالة. الدالة `triple` تتلقّى متغيرا من نوع `int` يسمى `number`.

السطر الذي يشير إلى أن الدالة تقوم بـ"إرجاع قيمة" هو السطر الذي يحتوي على الكلمة المفتاحية `return`. هذا السطر يوجد في العادة في نهاية الدالة، بعد الحسابات.

```
return result;
```

هذه الشفرة المصدرية تعني للدالة : "توقّفي و أرجعي العدد `result`". <u>يجب</u> أن يكون هذا المتغير `result` من نوع `int`، لأن الدالة تقوم بإرجاع قيمة من نوع `int` كما قلت في الأعلى.

صرّحت عن (= أنشأت) المتغير `result` في الدالة `triple`. هذا يعني أنه لا يستخدم إلا داخل هذه الدالة و ليس داخل أخرى كالدالة `main` مثلا. أي أنه متغير خاص بالدالة `triple`.

لكن هل هذه هي الطريقة الأقصر لكتابة الدالة `triple` ؟
لا، يمكننا كتابة محتوى الدالة في سطر واحد كالتالي :

```
int triple (int number)
{
        return number □ 3;
}
```

هذه الدالة تقوم بنفس المهمة التي تقوم بها الدالة السابقة، هي فقط أسرع من ناحية كتابتها. عموماً، الدوال التي تكتبها تحتوي الكثير من المتغيرات من أجل إجراء الحسابات عليها، نادرة هي الدوال القصيرة مثل `triple`.

العديد من المعاملات، لا معاملات

العديد من المعاملات

الدالة `triple` تحوي معاملاً واحداً، لكن من الممكن إنشاء دوال تقبل العديد من المعاملات.
مثلا، دالة `addition` لجمع عددين `a` و `b` :

```
int addition (int a, int b)
{
        return a + b;
}
```

يكفي تفريق المعاملات بفاصلة كما ترى.

لا معاملات

بعض الدوال، نادرة أكثر، لا تأخذ أية معامل كقيمة إدخال. هذه الدوال تقوم بنفس الشيء في غالب الأحيان. في الواقع، إذا لم تكن لديها أعداد تعمل عليها، فمهمة هذه الدوال هي القيام بوظائف معينة، كإظهار رسالة على الشاشة. و أيضاً، سيكون نفس النص الذي تظهره في كلّ مرة لأن الدالة لا تتلقّى أيّ معامل قد يكون قادرا على تغيير سلوكها !

تخيل دالة `hello` تقوم بإظهار الرسالة "Hello" على الشاشة:

```
void hello ()
{
        printf("Hello");
}
```

لم أضع أي شيء داخل الأقواس لأن الدالة لا تتلقّى أي معامل.
بالإضافة إلى ذلك، استعملت النوع `void` الذي كلّمتك عنه أعلاه.

بالفعل، كما ترى فالدالة لا تحتاج إلى التعليمة `return` لأنها لا ترجع أي شيء. الدالة التي لا ترجع أي شيء هي دالة من النوع `void`.

استدعاء دالة

سنقوم الآن بتجريب الشفرة المصدرية للتمرّن قليلاً مع ما تعلّمناه.
سنستعمل الدالة `triple` لضرب عدد في 3.

لحد الآن، أطلب منك كتابة الدالة `triple` قبل الدالة `main`. فإذا وضعتها بعدها، فلن يشتغل البرنامج. سأشرح لك هذا لاحقاً.

إليك الشفرة المصدرية التالية :

```
#include <stdio.h>
#include <stdlib.h>
int triple(int number)
{
        return 3 * number;
}
int main(int argc, char *argv[])
{
```

```
9         int inputNumber = 0, tripleNumber = 0;
10
11        printf("Enter a number... ");
12        scanf("%d", &inputNumber);
13
14        tripleNumber = triple(inputNumber);
15        printf("The number's triple = %d\n", tripleNumber);
16
17        return 0;
18 }
```

يبدأ البرنامج بالدالة `main` كما تعلم.
نطلب من المستعمل إدخال عدد. نبعث هذا العدد كإدخال للدالة `triple`، ثم نسترجع النتيجة في المتغير `tripleNumber`.
أنظر بشكل خاص السطر التالي:

```
1 tripleNumber = triple(inputNumber);
```

داخل القوسين، نبعث المتغير كمدخل للدالة `triple`، إنه العدد الذي ستعمل به الدالة. هذه الدالة تقوم بإرجاع قيمة، هذه القيمة نسترجعها في المتغير `tripleNumber`. نأمر إذا الحاسوب في هذا السطر : "أُطلب من الدالة `triple` ضرب العدد `inputNumber` في 3 و تخزين النتيجة في المتغير `tripleNumber`."

نفس الشرح على شكل مخطط

ألازالت لديك صعوبات في فهم المبدأ الذي تعمل به الدالة ؟
لا تقلق ! أنا متأكد أنك ستفهم بالمخططات.

هذه الشفرة المصدرية التي تحتوي على تعليقات سَتُريك في أي ترتيب يتم تنفيذ التعليمات. إبدأ إذا بقراءة السطر المرقّم 1 ثم 2، إلخ (أعتقد أنك فهمت) :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int triple(int number) // 6
4  {
5         return 3 * number; // 7
6  }
7  int main(int argc, char *argv[]) // 1
8  {
9         int inputNumber = 0, tripleNumber = 0; // 2
10
11        printf("Enter a number... "); // 3
12        scanf("%d", &inputNumber); // 4
13
14        tripleNumber = triple(inputNumber); // 5
15        printf("The number's triple = %d\n", tripleNumber); // 8
16
17        return 0; // 9
18 }
```

إليك ما يحدث سطراً بسطر :

1. يبدأ البرنامج من الدالة `main` .
2. يقرأ التعليمات في الدالة واحدة تلو الأخرى بالترتيب.
3. يقرأ التعليمة الخاصة بالدالة `printf` .
4. يقرأ أيضاً التعليمة الخاصة بالدالة `scanf` .
5. يقرأ التعليمة ... آه نحن نستدعي الدالة `triple` ، يجب إذا أن نقفز إلى أول سطر من محتوى هذه الدالة في الأعلى.
6. نقفز إلى الدالة `triple` ثم نقوم باسترجاع المعامل `number` .
7. نقوم بالحسابات و ننتهي من الدالة. الكلمة المفتاحية `return` تعني أن الدالة قد انتهت و تسمح بتحديد النتيجة التي سترجعها الدالة.
8. نرجع للدالة `main` إلى التعليمة الموالية.
9. نصل إلى `return` و منه تنتهي الدالة `main` و ينتهي البرنامج.

إذا فهمت في أي ترتيب يتم تنفيذ التعليمات فيه، فقد فهمت المبدأ. الآن يجب أن تفهم بأن الدالة تستقبل معاملات كمداخل و ترجع قيمة كمخرج.

```
#include <stdio.h>
#include <stdlib.h>

int triple(int number)
{
    return 3 * number;
}

int main(int argc, char *argv[])
{
    int inputNumber = 0, tripleNumber = 0;

    printf(" Enter a number... ");
    scanf("%d", &inputNumber);
    tripleNumber = triple(inputNumber);
    printf("The triple of this number= %d\n",tripleNumber);

    return 0;
}
```

1- يبعث المتغير **inputNumber** كمعامل للدالة **triple** تسترجع الدالة المتغير في متغير آخر **number**

2- ترجع الدالة **triple** قيمة تساوي ثلاث مرات العدد الذي تم إدخاله

يتم تخزين قيمة الإرجاع في متغير يسمى **tripleNumber** في الدالة الرئيسية تسمح الاشارة "=" بتخزين النتيجة في المتغير

ملاحظة : كان من الممكن أن أستعمل نفس اسم المتغير في الدالتين دون أي مشكل تداخل لأن كل متغير ينتمي لدالته الخاصة .

ملاحظة : ليس الأمر نفسه بالنسبة لكل الدوال. أحياناً، لا تأخذ الدالة أية معامل كإدخال، و بالعكس أحياناً تأخذ الكثير من المعاملات كإدخال (لقد شرحت لك هذا سابقاً).
أيضاً، يمكن لدالة أن ترجع قيمة كما يمكنها ألا ترجع أي شيء (و في هذه الحالة لا يكون هناك `return`).

فلنجرب هذا البرنامج

هذا مثال عن تنفيذ البرنامج :

```
Enter a number... 10
The number's triple = 30
```

> لست مضطراً إلى أن تخزن النتيجة في متغير ! يمكنك أن تعطي النتيجة المُرجعة من طرف الدالة `triple` إلى دالة أخرى و كأن التعليمة `triple(inputNumber)` في حدّ ذاتها متغير.

لاحظ هذا جيداً، هي نفس الشفرة المصدرية لكن هناك تغيير آخر على مستوى `printf` . بالإضافة إلى ذلك، لم نقم بالتصريح عن المتغير `tripleNumber` و الذي لا يفيدنا في أي شيء الآن :

```c
#include <stdio.h>
#include <stdlib.h>
int triple(int number)
{
        return 3 * nombre;
}
int main(int argc, char *argv[])
{
        int inputNumber = 0, tripleNumber = *0

        printf("Enter a number... ");
        scanf("%d", &inputNumber);
        // The result returned by the function is directly sent to printf
            without being saved in a variable
        printf("The number's triple = %d\n", triple(inputNumber));

        return 0;
}
```

كما ترى، استدعاء الدالة `triple` يتم داخل الدالة `printf` .
ماذا يفعل الجهاز حينما يصل إلى هذا السطر من الشفرة المصدرية ؟

الأمر سهل، يجد أن السطر يبدأ بـ `printf` ، فسيقوم إذا باستدعاء الدالة `printf` . يبعث إلى هذه الأخيرة كل المعاملات التي كتبناها. أول معامل هو النص الذي نريد طباعته و الثاني هو عدد.
يجد الجهاز بأنه قبل أن يبعث عددا إلى الدالة `printf` عليه أولا استدعاء الدالة `triple` . هذا ما يقوم به : يستدعي `triple` ، يقوم بالحسابات و ما إن يتلقّ النتيجة حتّى يبعثها للدالة `printf` !

هذه الحالة تمثّل نوعاً ما تداخل الدوال. الشيء الذي نستنتجه من هذا هو أنه بإمكان دالة أن تستدعي دالة أخرى ! هذا هو مبدأ البرمجة بلغة C ! كلّ شيء مركّب مع الأشياء الأخرى، كما في لعبة Lego.

في النهاية، سيبقى الشيء الأصعب هو كتابة الدوال. ما إن تكتبها، لن يبق عليك سوى استدعائها دون أن تلقي بالا على العمليات التي تجري بداخلها. هذا سيسمح لك بتبسيط كتابة برامجك بشكل كبير. و صدّقني ستحتاج إلى هذه المبادئ كثيراً !

2.9 أمثلة للفهم الجيّد

كان عليك أن تلاحظ شيئاً : أنا شخص يلحّ كثيراً على الأمثلة.
المفاهيم النظرية مهمّة، لكنها لا تكفي لتذكّر كلّ شيء، كما أنك لن تفهم في أي شيء يمكنك استغلالها لاحقاً. و هذا سيكون أمرا مؤسفا.

سأريك إذا الآن عدة استعمالات للدوال لكي تأخذ فكرة عن أهميّتها. سأقدّم الكثير من الأمثلة المتخلفة محاولاً أن أعطيك لمحة عن كلّ أنواع الدوال التي يمكن أن توجد.

لن أعلّمك أي شيء جديد، لكنّه قد حان الوقت لرؤية أمثلة تطبيقية. إذا كنت قد فهمت ما سبق فهذا أمر جيّد و لن يعيقك أي مثال مما سيأتي.

التحويل من الأورو الى الفرنك

سنبدأ بدالة مشابهة كثيراً للدالة `triple`، لكنها تحمل أهمية لابأس بها هذه المرة : دالة تحوّل من الأورو إلى الفرنك. لمن لا يعرف فإن 1 أورو = 6,55957 فرنك.

سننشئ دالة نسميها `conversion`.
هذه الدالة تأخذ متغيرا كإدخال من نوع `double` لأننا سنتعامل بالضرورة مع أعداد عشرية. إقرأها بتمعّن :

```
1  double conversion(double euros)
2  {
3          double francs = 0;
4          francs = 6.55957*euros;
5          return francs;
6  }
7  int main(int argc, char * argv[])
8  {
9          printf("10 euros = %fF\n", conversion(10));
10         printf("50 euros = %fF\n", conversion(50));
11         printf("100 euros = %fF\n", conversion(100));
12         printf("200 euros = %fF\n", conversion(200));
13         return 0;
14 }
```

```
10 euros = 65.595700F
50 euros = 327.978500F
100 euros = 655.957000F
200 euros = 1311.914000F
```

لا يوجد اختلاف كبير مقارنة بالدالة `triple`، لقد أخبرتك بذلك مسبّقاً. الدالة `conversion` طويلة قليلاً و يمكن أن يتم اختصارها في سطر واحد، سأترك لك عناء فعل ذلك بنفس الطريقة التي شرحتها لك مسبقاً.

في الدالة `main`، تعمّدت وضع الكثير من `printf` لأريك الهدف من استعمال الدوال. لكي أحصل على قيمة 50 أورو، ليس علي سوى استدعاء الدالة `conversion` بإعطائها 50 كقيمة إدخال. و إذا أردت تحويل 100 أورو إلى الفرنك، كل ما أحتاج إلى فعله هو تغيير المعامل المرسل إلى الدالة (وضع القيمة 100 في مكان القيمة 50).

حان دورك ! اكتب دالة ثانية (دائماً قبل الدالة `main`) تقوم بالعملية العكسية أي تحول من الفرنك إلى الأورو. لن يكون الأمر صعباً. هناك إشارة عملية تتغير ليس إلا.

العقوبة

سنهتم الآن بدالة لا تقوم بإرجاع أي شيء (لا وجود للإخراج).
هي دالة تقوم بإظهار نفس النص على الشاشة بالقدر الذي نحن نريد. هذه الدالة تأخذ كإدخال : عدد المرات التي نريد أن يظهر بها نص العقوبة على الشاشة.

```
void punishment(int numberOfLines)
{
        int i;
        for (i=0; i<numberOfLines; i++){
                printf("I won't misbehave in class again\n");
        }
}
int main(int argc, char * argv[])
{
        punishment(10);
        return 0;
}
```

```
I won't misbehave in class again
I won't misbehave in class again
I won't misbehave in class again
I won't misbehave in class again
I won't misbehave in class again
I won't misbehave in class again
I won't misbehave in class again
I won't misbehave in class again
I won't misbehave in class again
I won't misbehave in class again
```

هنا نتكلم عن دالة لا تُرجع أية قيمة. تكتفي هذه الدالة بالقيام بمهمة (هنا، إظهار النص على الشاشة).
الدالة التي لا ترجع أيه قيمة هي دالة من نوع `void` ، غير هذا لا يوجد شيء مختلف.

سيكون من الممتع أن نكتب دالة `punishment` تتلائم مع أيّ عقوبة، فتقوم بإظهار النص الذي نريده نحن على الشاشة. نبعث لها معاملين : النص الذي نريد و عدد المرات التي نريد أن يتم إظهاره. المشكل هو أننا لا نجيد بعد التعامل مع النصوص في الـC (إذا كنت لم تنتبه فأذكرك أنه لحدّ الآن لم نتعامل سوى مع متغيرات تحمل قيما عدديّة داخلها منذ بداية الكتاب !). و بهذا الصدد، أخبرك أننا لن نتأخر حتى نتعلّم كيفية التعامل مع النصوص. فعل ذلك معقّد قليلاً و لا يصلح أن نبدأ به في أول الكتاب !

مساحة مستطيل

من السهل حساب مساحة المستطيل : العرض × الطول.
الدالة التي سنكتبها `rectangleSurface` تأخذ معاملين : الطول و العرض. و تقوم بإرجاع المساحة.

```
double rectangleSurface(double width, double height)
{
        return width * height;
}
int main(int argc, char * argv[])
{
        printf("Width = 5 and height = 10. Surface = %f\n", rectangleSurface
            (5,10));
        printf("Width = 2.5 and height = 3.5. Surface = %f\n", rectangleSurface
            (2.5,3.5));
        printf("Width = 4.2 and height = 9.7. Surface = %f\n", rectangleSurface
            (4.2,9.7));
        return 0;
}
```

```
Width = 5 and height = 10. Surface = 50.000000
Width = 2.5 and height = 3.5. Surface = 8.750000
Width = 4.2 and height = 9.7. Surface = 40.740000
```

؟

هل يمكننا أن نظهر مباشرة طول، عرض و مساحة المستطيل داخل الدالة ؟

بالطبع ! في هذه الحالة لن ترجع الدالة أي شيء. ستكتفي بإظهار ما حسبته :

```
void rectangleSurface(double width, double height)
{
        double surface;
        surface = width * height;
        printf("Width = %f and height = %f. Surface = %f\n", width, height,
            surface);
}
int main(int argc, char * argv[])
{
        rectangleSurface(5,10);
        rectangleSurface(2.5,3.5);
        rectangleSurface(4.2,9.7);
        return 0;
}
```

كما ترى، الـ `printf` في داخل الدالة `rectangleSurface` يعرض نفس الرسالة السابقة. هذه فقط طريقة مختلفة لفعل نفس الشيء.

القائمة

هذه الشفرة أكثر أهمية و واقعيّة. سننشئ دالة `menu` لا تأخذ أي معامل كإدخال. تكتفي هذه الدالة بإظهار قائمة و تطلب من المستعمل اختيار ما يريد. الدالة تقوم بإرجاع اختيار المستعمل.

```
int menu()
{
        int choice = 0;

        while (choice < 1 || choice > 4)
        {
                printf("Menu :\n");
                printf("1 : Royal Fried Rise\n");
                printf("2 : Noodle Basil \n");
                printf("3 : Pattaya Paradise \n");
                printf("4 : Spice Nitan Spicy...\n");
                printf("Your choice? ");
                scanf("%d", &choice);
        }

        return choice;
}
int main(int argc, char * argv[])
{
        switch (menu())
        {
                case 1:
                printf("You have chosen Royal Fried Rise\n");
                break;
                case 2:
                printf("You have chosen Noodle Basil \n");
                break;
                case 3:
                printf("You have chosen Pattaya Paradise \n");
                break;
                case 4:
                printf("You have chosen Spice Nitan Spicy\n");
                break;
        }
        return 0;
}
```

اغتنمت الفرصة لتحسين القائمة (مقارنة بما نفعله عادة) : تقوم الدالة `menu` بإظهار القائمة طالما لم يقم المستعمل بإدخال رقم محصور بين 1 و 4. فهكذا، لا يمكن أن تقوم الدالة بإرجاع قيمة لا تظهر على القائمة !

في الـ `main`، تلاحظ أننا استعملنا `switch(menu())` . بمجرّد أن تنتهي `menu`، فإنّها ستعيد خيار المستخدم مباشرة في `switch` . إنها طريقة سريعة و عمليّة.

حان دورك ! يمكنك تحسين الشفرة المصدرية أكثر : يمكنك أن تظهر رسالة خطأ في حالة أخطأ المستعمل في الإختيار بدل أن تقوم بإعادة إظهار القائمة على الشاشة مرّة أخرى.

ملخّص

- يمكن لدالة أن تستدعي دالة أخرى. و لهذا فالـ `main` يمكنها استدعاء دوال جاهزة كـ `printf` و `scanf` و كذلك دوالا ننشؤها بأنفسنا.
- تتلقّى الدالة كإدخال متغيّرات نسميها **معاملات.**
- تقوم الدالة ببعض العمليّات على هذه المعاملات و تُرجع عادة قيمة بالاستعانة بالتعليمة `return`.

الجزء ب

تقنيات متقدّمة في لغة الـC

الفصل 10

البرمجة المجزّأة (Modular Programming)

في هذه المرحلة الثانية، سنكتشف مبادئ متقدّمة في لغة الـC لن أخفي عليك، هذه المرحلة صعبة الفهم و تحتاج منك التركيز. في نهاية المرحلة، ستكون قادراً على تدبّر أمرك في معظم البرامج المكتوبة بلغة الـC. في المرحلة التي تليها نتعلّم كيف نفتح نافذة، كيف ننشئ لعبة ثنائية الأبعاد... إلخ.

لحدّ الآن عملنا في ملف واحد سمّيناه `main.c` . كان أمراً مقبولاً لحدّ الآن لأن براجمنا كانت صغيرة، لكنها ستصبح في القريب العاجل مركّبة من عشرات، لن أقول من مئات الدوال، و إن كنت تريد وضعها كلّها في نفس الملف، فإن هذا الأخير سيصبح ضخماً جداً. لهذا السبب تم اختراع ما نسمّيه بالبرمجة المجزّأة. المبدأ سهل : بدل أن نضع كل الشفرة المصدرية في ملف واحد `main.c` ، سنقوم بتفريقها إلى عدة ملفات.

1.10 النماذج (Prototypes)

لحدّ الآن، كنت عندما تنشئ دالة، أطلب منك وضعها قبل الدالة الرئيسية `main` . لماذا ؟

لأن للترتيب أهمية حقيقية هنا : فإن قمت بوضع الدالة قبل الـ`main` في الشفرة المصدرية، سيقرؤها الجهاز و يتعرف عليها. حينما تقوم باستدعاء الدالة داخل الـ`main` ، سيعرفها الجهاز و يعرف أيضاً أين يبحث عليها.
بالعكس، لو تضع الدالة بعد الـ`main` ، لن يعمل البرنامج لأن الجهاز لم يتعرّف بعد على الدالة. جرّب ذلك و سترى.

> ؟ لكنه تصميم سيّء نوعاً ما، أليس كذلك ؟

أنا متفق معك ! لكن انتبه المبرمجون لهذه النقطة قبلك و عملوا على حلّ المشكل.

بفضل ما سأعلمك إياه الآن، ستتمكن من الدوال في أي ترتيب كان في الشفرة المصدرية، هكذا لن تقلق من هذه الناحية.

استعمال النموذج للتصريح عن دالة

سنقوم بتصريح دوالنا للحاسوب، و هذا بكتابة ما نسميه بالـ**نماذج**. لا تنبهر بهذا الاسم، إنه يخبّئ معلومة بسيطة جداً.

تأمل في السطر الأول من دالتنا `rectangleSurface`

```
double rectangleSurface(double width, double height)
{
        return width * height;
}
```

قم بنسخ السطر الأول (`double rectangleSurface...`) المتواجد أعلى الشفرة المصدرية (مباشرة بعد تعليمات التضمين `#include`). أضف فاصلة منقوطة في نهاية هذا السطر.
و هكذا يمكنك أن تضع الدالة الخاصة بك `rectangleSurface` بعد الدالة `main` ان أردت !

هذا ما يجب أن تكون عليه الشفرة المصدرية :

```
#include <stdio.h>
#include <stdlib.h>
// The next line represents the prototype of the function rectangleSurface :
double rectangleSurface(double width, double height);
int main(int argc, char *argv[])
{
        printf("width = 5 and height = 10. Surface = %f\n", rectangleSurface(5,
            10));
        printf("width = 2.5 and height = 3.5. Surface = %f\n", rectangleSurface
          (2.5, 3.5));
        printf("width = 4.2 and height = 9.7. Surface = %f\n", rectangleSurface
          (4.2, 9.7));

        return 0;
}
// Now, we can put our function wherever we want in the source code :
double rectangleSurface(double width , double height )
{
        return width * height ;
}
```

الشيء الذي تغيّر هنا هو إضافة النموذج أعلى الشفرة المصدرية.
النموذج هو عبارة عن إشارة للجهاز، يوحي إليه بوجود دالة تسمى `rectangleSurface` و التي تأخذ معاملات إدخال معينة و تُرجِع مخرجا من نوع أنت من تحدده. هذا يساعد الجهاز على تنظيم نفسه.

بفضل ذلك السطر، يمكنك الآن وضع دوالك في أي ترتيب كان دون أي تفكير زائد.

أكتب دائما النموذج الخاص بدوالك. البرامج التي ستكتبها من الآن و صاعداً ستصبح أكثر تعقيداً و تستعمل الكثير من الدوال : من الأحسن أن تتعلّم منذ الآن العادة الجيدة بوضع نموذج لكل دالة في الشفرة المصدرية.

كما ترى، الدالة `main` لا تملك أي نموذج، و كمعلومة فهي الوحيدة التي لا تملك نموذجاً ! لأن الجهاز يعرفها (فهي نفسها مكررة في جميع البرامج).

عليك أن تعرف أنه في سطر النموذج، لست مضطراً إلى تحديد المعاملات التي تتلقاها الدالة كمدخل. الجهاز يحتاج أن يتعرّف إلى نوع المداخل فقط.

يمكننا أن نكتب ببساطة :

```
1 double rectangleSurface (double, double);
```

و مع ذلك، فالطريقة التي أريتك إياها أعلاه تعمل أيضاً. الشيء الجيد فيها هو أن كلّ ما عليك فعله هو نسخ و لصق السطر الأول الخاص بالدالة مع إضافة فاصلة منقوطة (طريقة سهلة و سريعة).

> X
>
> لا تنس أبدا وضع فاصلة منقوطة بعد النموذج، هذا يمكّن الحاسوب من التفريق بين النموذج و بداية الدالة. إن لم تفعل، ستعترضك أخطاء غير مفهومة أثناء عملية الترجمة.

2.10 الملفات الرأسية (Headers)

لحدّ الآن لا نملك غير ملفٍ مصدري واحد في مشروعنا و هو الذي كنّا نسمّيه `main.c`.

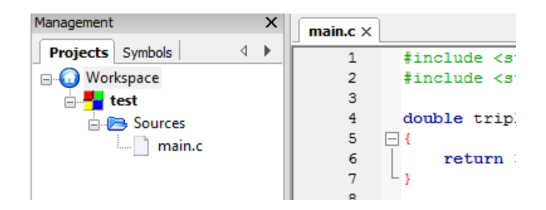
عدة ملفات في مشروع واحد

تطبيقياً، برامجك لن تكون مكتوبة في ملف واحد `main.c`. بالطبع يمكن فعل ذلك، لكن لن يكون من الممتع أن تتجوّل في ملف به 10000 سطر (شخصياً أعتقد هذا). و لهذا فإنه في العادة ننشئ العديد من الملفات في المشروع الواحد.

> ؟
>
> عفوا ... ماهو المشروع ؟

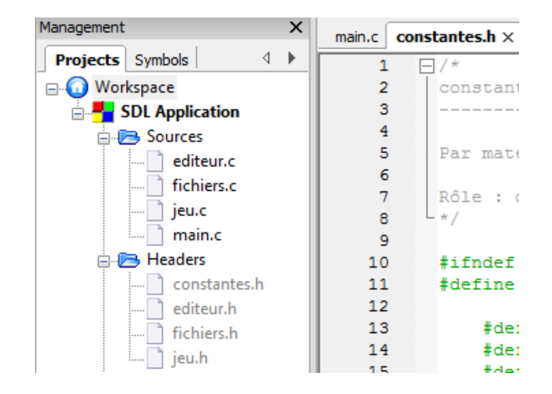
لا ! هل نسيت بسرعة ؟ سأعيد الشرح لأنه من اللازم أن نتّفق على هذا المصطلح.

المشروع هو مجموع الملفات المصدرية الخاصة ببرنامجك. لحد الآن برنامجنا لم تتكون إلا من ملف واحد. و يمكنك التحقق من هذا بالنظر في البيئة التطويرية الخاصة بك، غالبا ما يظهر المشروع في القائمة على اليسار (الصورة الموالية) :



كما يمكنك رؤيته في يسار الصورة، هذا المشروع ليس مكوّنا إلا من الملف `main.c`.

اسمح لي الآن أن أُرِيَكَ صورة لمشروع حقيقي ستقوم به في وقت لاحق من الكتاب : لعبة Sokoban :



كما ترى، هناك ملفات عديدة. هذا ما يكون عليه المشروع الحقيقي، أي نتواجد به ملفات عديدة في القائمة اليسارية يمكن التعرّف على الملف `main.c` من بين القائمة و الذي يحتوي الدالة `main`. بصورة عامة في برامجي، لا أضع إلّا الدالة `main` في الملف `main.c`. لمعلوماتك، هذا ليس أمراً إجبارياً، كل واحد ينظّم ملفاته بالشكل الذي يريد. لكن لكي تتبعني جيّداً أنصحك بفعل ذلك.

> ؟ لكن لم يجب عليّ إنشاء ملفات عديدة ؟ و كم من ملف يجب عليّ أن أنشئ في مشروعي ؟

هذا يبقى اختيارك أنت، في الغالب نجمع في نفس الملف المصدري الدوال التي تشترك في الموضوع الذي تعالجه. و هكذا ففي الملف `editeur.c` جمعت كلّ الدوال الخاصة ببناء المستوى، و في الملف `jeu.c` قمت بتجميع الدوال الخاصة باللعبة نفسها و هكذا ...

الملفات `.c` و `.h`

كما يمكنك أن تلاحظ، يوجد نوعان مختلفان من الملفات في الصورة السابقة.

- **ملفات ذات الإمتداد `.c`** : الملفات المصدرية، تحتوي الدوال نفسها.
- **ملفات ذات الإمتداد `.h`** : تسمى الملفات الرأسية و هي تحتوي النماذج الخاصة بالدوال.

عموما، إنه لمن النادر وضع نماذج في الملفات من صيغة `.c` مثلما فعلنا للتوّ في الملف `main.c` (إلا إذا كان برنامجك صغيرا).

من أجل كل ملف `.c` هناك ملف مكافئ له، و الذي يحتوي نماذجا للدوال الموجودة في الملف `.c`، تمعّن في الصورة السابقة.

- هناك `editeur.c` (الشفرة الخاصة بالدوال) و `editeur.h` (ملف النماذج الخاصة بالدوال).
- هناك `jeu.c` و `jeu.h`.
- إلخ.

لكن كيف يعرف الحاسوب بأن نماذج الدوال موجودة في ملف آخر خارج الملف `.c` ؟

يجب عليك تضمين الملف الرأسي `.h` . مستعيناً بتوجهات المعالج القبلي.
كن مستعداً لأنّي سأعطيك الكثير من المعلومات في وقت قصير.

كيف نقوم بتضمين ملف رأسي ؟ أنت تجيد فعل ذلك لأنك قمت بذلك من قبل.

أنظر مثالاً من بداية الملف `jeu.c` :

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include "jeu.h"
4 void play(SDL_Surface* screen)
5 {
6 // ...
```

التضمين يتم عن طريق توجيهات المعالج القبلي `#include` التي يجدر بك أن تكون قد تعلّمتها من قبل.
تمعن في التالي :

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include "jeu.h" // We include jeu.h
```

قمنا بتضمين ثلاثة ملفات من صيغة `.h` و هي : `stdio`، `stdlib` و `jeu`.
لاحظ الفرق : الملفات التي قمت بإنشائها ووضعها في المجلّد الخاص بمشروعك يجب أن تكون مضمّنة بإشارات الاقتباس (`"jeu.h"`) بينما ملفات المكتبات (التي توجد عادة في البيئة التطويرية الخاصة بك) تكون مضمّنة بعلامات الترتيب (`<stdio.h>`).

تستعمل إذا :

- علامتي الترتيب `< >` : لتضمين الملفات المتواجدة في المجلّد `include` الخاص بالبيئة التطويرية.
- علامتي الاقتباس `" "` : لتضمين الملفات المتواجدة في مجلّد المشروع (و غالبا بجانب الملفات `.c`).

الأمر `#include` يطلب إدخال محتوى ملف معيّن في الملف `.c` فهي تعليمة تقول: "أدخل الملف `jeu.h` هنا" مثلا .

و في الملف `jeu.h` ماذا نجد ؟

لا نجد إلا نماذج خاصة بدوال الملف `jeu.c` !

```c
void play(SDL_Surface* screen);
void movePlayer(int map[][NB_BLOCS_HEIGHT], SDL_Rect *pos, int direction);
void moveBox(int *firstBox, int *secondeBox);
```

هكذا يعمل المشروع الحقيقي !

؟ ما الهدف من وضع نماذج في ملفات من نوع `.h` ؟

السبب بسيط للغاية، عندما تستدعي دالة في الشفرة المصدرية الخاصة بك، ينبغي لجهازك أن يكون متعرفا عليها من قبل، و يعرف كم من المعاملات تستعمل...إلخ. إن هذا هو الهدف وراء وجود النماذج، إنه دليل الاستخدام الخاص بالدالة بالنسبة لجهاز.

كلّ هذا هو مسألة تنظيم، عندما تضع نماذجك في ملفات `.h` (ملفات رأسية) مضمّنة في أعلى الملفات `.c`، سيعرف جهازك طريقة استخدام الدوال الموجودة في الملف ما إن يبدأ في قراءته.

عند القيام بهذا، لن يكون عليك القلق حيال الترتيب الذي ستكون عليه دوالك في الملفات `.c`. إذا كنت قمت الآن بإنشاء برنامج صغير يحتوي على دالتين أو ثلاث يمكنك أن تفكّر أنه من الممكن للبرنامج أن يشتغل دون وجود النماذج، لكن هذا لن يستمر طويلا ! فما إن يكبر البرنامج و إن لم تنظّم النماذج في ملفات رأسيّة فستفشل الترجمة دون أدنى شك.

م عندما تستدعي دالة متواجدة في الملف `functions.c` إنطلاقا من الملف `main.c` سيكون عليك تضمين النماذج الخاصة بالملف `functions.c` في الملف `main.c` يجب إذن وضع `#include "functions.h"` في أعلى الملف `main.c`.
تذكّر هذه القاعدة : "في كلّ مرة تستدعي الدالة X في ملف، يجب عليك إدراج نموذج هذه الدالة في ملفك" هذا ما يسمح للمترجم بمعرفة ما إن كنت قد استدعيتها بشكل صحيح.

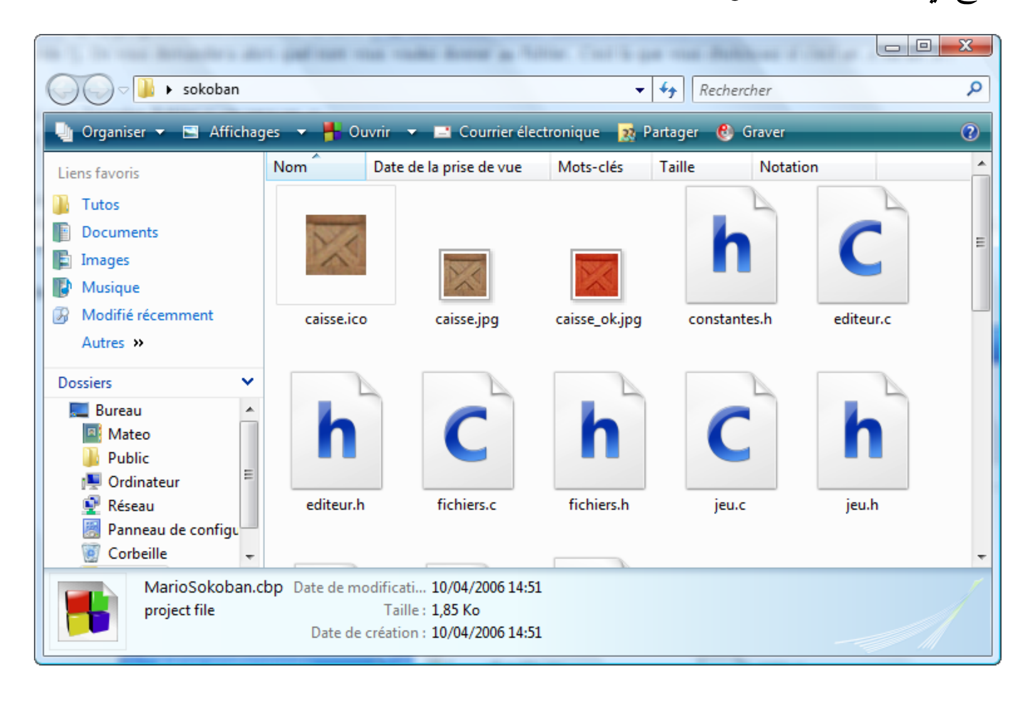
؟ كيف أقوم بإضافة ملفات `.c` و `.h` إلى مشروعي ؟

هذا راجع للبيئة التطويرية التي تستخدمها. لكن المبدأ هو نفسه في جميع البرامج : `File` / `New` / `Source File` هذا يسمح بإنشاء ملف جديد فارغ. هذا الملف ليس حاليا من النوع `.c` ولا `.h` أنت من يحدد ذلك أثناء عملية حفظ الملف. قم إذن بحفظه (حتّى و إن كان لا يزال فارغا !) و هنا يطلب منك إدخال اسم للملف، يمكنك هنا اختيار صيغة الملف :

- إذا سميته `file.c` فسيكون بامتداد `.c`.
- إذا سميته `file.h` فسيكون بامتداد `.h`.

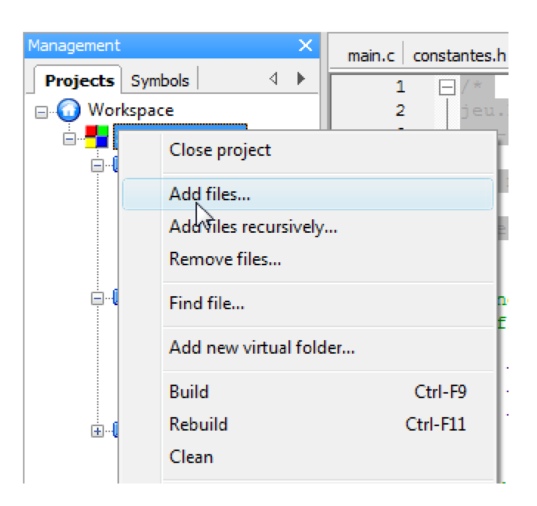
هذا سهل. قم بحفظ الملف في المجلّد أين تتواجد باقي الملفات الخاصة بمشروعك (نفس المجلّد أين يتواجد الملف `main.c`). عموما كل ملفات المشروع تقوم بحفظها في نفس المجلّد سواء كانت ذات صيغة `.c` أو `.h`.

مجلّد المشروع في النهاية سيكون مثل هذا :



الملف الذي أنشأته محفوظ لكن لم تتم إضافته إلى مشروعك بعد !

لإضافته قم بالنقر يمينا على القائمة أيسر الشاشة (الخاصة بملفات المشروع) و اختر `Add files` كالتالي :



ستظهر لك نافذة تطلب منك اختيار الملفات التي تريد أن تدخلها للمشروع، اختر الملف الذي قمت بإنشاءه، للتو، و سيتم إدخاله أخيرا في المشروع. ستجده حاضراً في القائمة اليسارية !

الـinclude الخاصّة بالمكتبات النموذجية

يفترض أنّ لديك سؤالا يدور في رأسك الآن...

إذا ضمّنا الملفات `stdio.h` و `stdlib.h` فهذا يعني أنهما موجودان في مكان ما و يمكننا البحث عنهما، أليس

كذلك ؟

نعم بالطبع !

يفترض أنهما مسطبان في المكان الذي تتواجد به البيئة التطويرية الخاصة بك، بالنسبة للبيئة Code::Blocks أجدهم هنا :

`C:\Program Files\CodeBlocks\MinGW\include`

على العموم يجب البحث عن مجلد يحمل اسم `include`.

بداخله تجد كمّا هائلا من الملفات، و هي ملفات رأسية (`.h`) خاصة بمكتبات نموذجية أي مكتبات متوفرة في كل مكان (سواء في Windows أو Mac OS X أو GNU/Linux...)، و ستجد داخلها الملفات `stdio.h` و `stdlib.h` مع ملفّات أخرى.

يمكنك فتحها إذا أردت، لكن ستتفاجئ بالعديد من الأشياء التي لم أدرّسها لك من قبل خاصة بما يتعلق ببعض توجيهات المعالج القبلي. يمكنك أن تلاحظ بأن الملف مليء بنماذج لدوال نموذجية مثل `printf`.

> حسناً، الآن عرفت أين أجد نماذج الدوال النموذجية لكن ألا يمكنني رؤية الشفرة المصدرية الخاصة بالدوال ؟ أين هي الملفات `.c` ؟

إنها غير موجودة أساسا ، لأنها مترجمة (إلى ملفات ثنائية (binary files)، يعني إلى لغة الحاسوب). و لهذا فإنه من المستحيل أن تقرأها.

يمكنك إيجاد الملفات المترجمة في المجلّد المسمى `lib` (و الذي هو اختصار لكلمة `library` أي مكتبة)، بالنسبة لي هي موجودة في المسار :

`C:\Program Files\CodeBlocks\MinGW\lib`

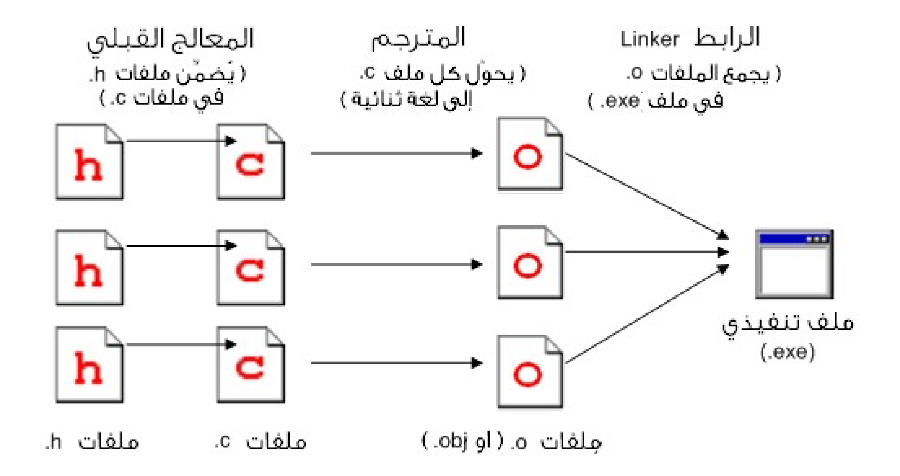
ملفات المكتبات المترجمة لها الصيغة `.a` في البيئة Code::Blocks و التي تستخدم `mingw` كمترجم. و لها صيغة `.lib` في برنامج Visual C++ الذي يستخدم المترجم `Visual`. لا تحاولوا قراءتها لأنها غير قابلة للقراءة من طرف إنسان عادي.

باختصار، يجب عليك أن تضمّن الملفات الرأسية `.h` في الملفات `.c` تتمكن من استخدام الدوال النموذجية مثل `printf` و كما تعرف فالجهاز على اطلاع على النماذج فهو يعرف ما إن كنت قد طلبت الدوال بشكل صحيح (إن لم تنس أحد المعاملات مثلا).

3.10 الترجمة المنفصلة

الآن و بعدما عرفت أن المشروع مبني على أساس ملفات مصدرية عديدة، يمكننا الدخول الآن في تفاصيل عملية الترجمة فلحد الآن لم نر سوى مخطط مبسط عنها.

سأعطيك الآن مخططا مفصلا عنها و من المستحسن أن تحفظه عن ظهر قلب :



هذا مخطط حقيقي عمّا يجري بالضبط أثناء التجميع و سأشرحه لك :

1. **المعالج القبلي** : المعالج القبلي هو برنامج ينطلق قبل عملية الترجمة و هو مخصص للقيام بتشغيل تعليمات نطلبها منه عن طريق ما سميناه بتوجيهات المعالج القبلي، و هي الأسطر الشهيرة التي تبدأ بإشارة `#`.

   لحد الآن توجيهة المعالج القبلي الوحيدة التّي نعرفها هي `#include` و التّي تسمح بإدراج ملف في ملف آخر. طبعا للمعالج القبلي مهام أخرى سنتعرف إليها لاحقا لكن ما يهمنا الآن هو ما أعطيتك إياه. المعالج القبلي يقوم إذن بـ"استبدال" أسطر `#include` بملفات أخرى نحددها، فهو يضمّن داخل كل الملفات `.c` الملفات `.h` التي نعينها و نطلب منه تضمينها في السابقة.

2. **الترجمة** : هذه الخطوة المهمة التي تسمح بتحويل ملفاتك إلى شفرات ثنائية مفهومة للحاسوب. فالمترجم يقوم بتجميع الملفات `.c` واحدا بواحدا حتى ينهيها جميعها، و لضمان ذلك يجب أن تكون كل الملفات موجودة في المشروع (بحيث تظهر في القائمة اليسارية).

   سيقوم المترجم بتوليد ملف `.o` أو `.obj` و هذا راجع لنوع المترجم و هي ملفات ثنائية مؤقتة، و على أي حال تحذف هذه الملفات في نهاية الترجمة و لكن بتعديل الخيارات يمكنك الإبقاء عليها لكن لكن يكون هناك من داع.

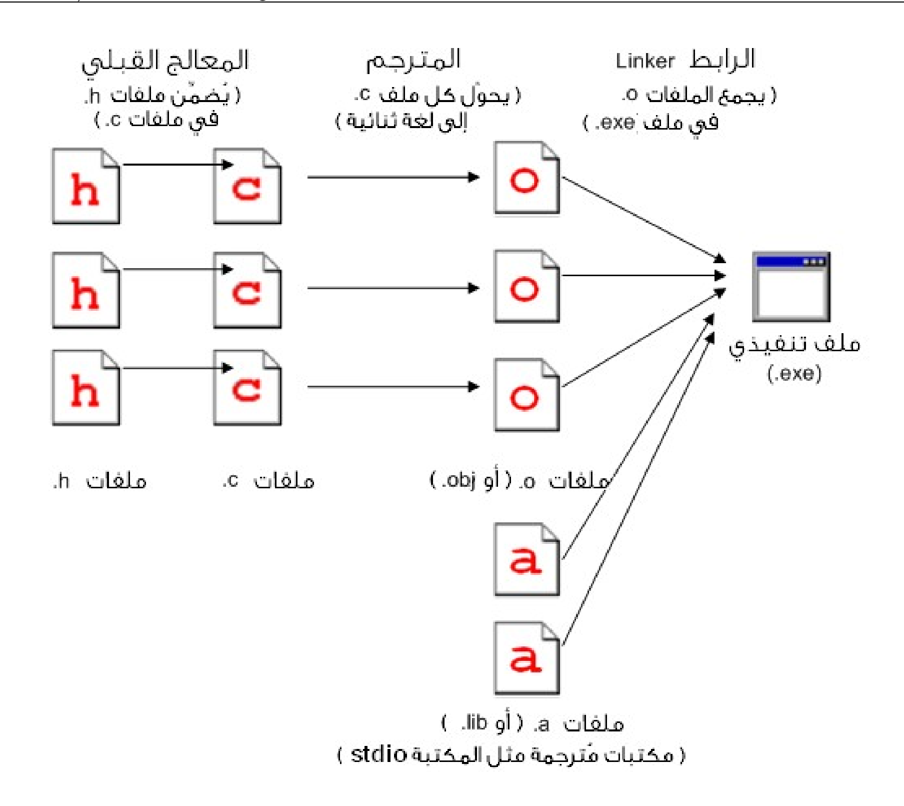
3. **إنشاء الروابط** : محرر الروابط (Linker) هو برنامج يعمل على جمع الملفات الثنائية من نوع `.o` في ملف واحد كبير : الملف التنفيذي النهائي ! هذا الملف يحمل الصيغة `.exe` في الويندوز. إن كنت تملك نظام تشغيل آخر فسيأخذ الصيغة المناسبة له.

و هكذا تكون قد تعرفت على الطريقة الحقيقية لعمل الترجمة. كما قلتها و أكررها المخطط أعلاه مهم للغاية، فهو يفرق بين مبرمج يقوم بجمع و نسخ الشفرة المصدرية دون فهم و بين مبرمج يعرف تماما ما عليه فعله !

معظم الأخطاء تحدث في الترجمة و قد تأتي من محرر الروابط و هذا يعني أنه لم يتمكن من تجميع كل الملفات `.o` بطريقة صحيحة (ربمّا لفقدان إحداها).

لا يزال المخطط أعلاه غير كامل، إذ أن المكتبات لم تظهر فيه ! إذن كيف تحدث العملية عندما نستخدم مكتبات برمجية ؟

تبقى بداية المخطط هي نفسها، لكن يقوم محرر الروابط بأعمال أخرى، سيقوم بتجميع ملفاتك `.o` (المؤقتة) مع مكتبات جاهزة تحتاجها (`.a` أو `.lib` وفقا للمترجم) :



هكذا نتهي و يكون مخططنا هذه المرة كاملا، ملفاتك من المكتبات `.a` (أو `.lib`) يتم تجميعها في الملف التنفيذي مع الملفات `.o`.

فبهذه الطريقة تحصل في النهاية على برنامج كامل 100% و الذي يحتوي كل التعليمات اللازمة للجهاز لتشرح له كيف يعرض نصّا !
كمثال، الدالة `printf` توجد في ملف `.a` و طبعا سيتم تجميعها مع الشفرة المصدرية الخاصّة بنا في الملف التنفيذي.

لاحقا سنتعلم كيف نستخدم المكتبات الرسومية التي نجدها أيضا في ملفات `.a` و تعطي للجهاز تعليمات خاصة بكيفية إظهار نافذة على الشاشة كمثال. لكن طبعا، لن ندرسها الآن ، كلّ شيء في وقته.

4.10 نطاق الدوال و المتغيرات

لننهي هذا الفصل، يجب أن أطلعكم عما يسمى **بنطاق** المتغيرات و الدوال، سنعرف إمكانية الوصول للدوال و المتغيرات، يعني متى يمكننا استدعاؤها.

المتغيرات الخاصة بدالة

عندما تصرّح عن متغير في داخل دالة يتم حذف هذا المتغير من الذاكرة مع نهاية الدالة.

```
int triple(int number)
{
        int result = 0; // The variable result is created in the memory
        result = 3 * number;
        return result;
} // The function finished, the variable result is destroyed
```

كلّ متغير تم التصريح عنه في دالة، لا يكون موجودا سوى حينما تكون الدالة في طور الإشتغال.
لكن ماذا يعني هذا تحديدا ؟ أنه لا يمكن الوصول إليه من خلال دالة اخرى !

```c
int triple(int number);
int main(int argc, char *argv[])
{
        printf("The triple of 15 = %d\n", triple(15));
        printf("The triple of 15 = %d",  result); // Error
        return 0;
}

int triple(int number)
{
        int result = 0;
        result = 3 * number;
        return result;
}
```

في الدالة الرئيسية أحاول الوصول إلى المتغير `result` و بما أن هذا المتغير تم التصريح عنه داخل الدالة `triple` فطبعا لا يمكنني الوصول إليه من خلال الدالة `main` !

تذكّر جيّدا : كل متغير تم التصريح عنه داخل دالة، لا يسرى مفعوله إلا في داخل هذه الدالة نفسها ! و نقول أن المتغير محلّي (Local).

المتغيرات الشاملة (Global variables) : فلتتجنّبها

متغير شامل قابل للوصول إليه من خلال كلّ الملفات

إنه من الممكن التصريح عن متغير يمكن الوصول إليه من خلال كل الدوال من ملفات المشروع. سأريك كيفية فعل ذلك كي تعرف بأنه أمر موجود، لكن عموما تجنب القيام بذلك. قد يظهر أنها ستسهل لك التعامل مع الشفرة المصدرية لكن قد يؤدي بك هذا لوجود العديد من المتغيرات التي يمكننا الوصول إليها من كلّ مكان ما سيصعّب عليك عملية إدارتها.

للتصريح عن متغير **شامل** (Global)، يجب أن تقوم بذلك خارج كلّ الدوال، يعني في أعلى الملف، و عموما بعد أسطر الـ `#include` .

```c
#include <stdio.h>
#include <stdlib.h>

int result = 0; // Declaration of a global variable
void triple(int number ); // Prototype of the function
int main(int argc, char *argv[])
{
        triple(15); // We call the function triple which is going to modify the
            variable result
```

```
9         printf("The triple of 15 = %d\n", result); // We can access to the
              variable result
10        return 0;
11  }
12
13  void triple(int number)
14  {
15        result = 3 * number;
16  }
```

في هذا المثال، الدالة `triple` لا تُرجع أي شيء (`void`). إنها تقوم بتعديل قيمة المتغير الشامل `result` التي يمكن للدالة `main` أن تسترجعه.

المتغير `result` يمكن الوصول إليه من خلال كل الملفات في المشروع و منه يمكننا استدعاؤها من خلال كلّ دوال البرنامج.

! هذا شيء يجب ألا يتواجد في برامج الـC الخاصة بك. من المستحسن استعمال التعليمة `return` لإرجاع النتيجة بدل التعديل عليه كمتغير شامل.

متغير شامل قابل للوصول إليه من خلال ملف واحد

المتغير الشامل الذي أريتك إياه قبل قليل يمكن الوصول إليه من خلال كل الملفات الخاصة بالمشروع.
يمكننا جعل متغيّر شامل مرئيا فقط في الملف الذي نتواجد به. و لكنه يبقى متغيرا شاملا على أية حال حتى و إن كنا نقول أنه ليس كذلك إلا على الدوال المتواجدة في ذات الملف و ليس على كل دوال البرنامج.

لإنشاء متغير شامل مرئي في ملف واحد نستعمل الكلمة المفتاحية `static` قبله :

```
1  static int result = 0;
```

متغير ساكن (static) بالنسبة لدالة

حذار : الأمر حساس هنا قليلاً. إن استعملت الكلمة المفتاحية `static` عند التصريح عن متغير في داخل دالة، فلهذا معنى آخر غير الخاص بالمتغيرات الشاملة.
في هذه الحالة، لا يتم حذف المتغير الساكن مع نهاية الدالة، بل حينما نستدعي الدالة مرّة أخرى، سيحفظ المتغير قيمته مثلا :

```
1  int triple(int number)
2  {
3        static int result = 0; // The first time when the variable is created
4        result = 3 * number;
5        return result;
6  } // When we exit the function, the variable is not destroyed
```

ماذا يعني هذا بالضبط ؟
يعني أنه يمكننا استدعاء الدالة لاحقا و يبقى المتغير `result` محتفظا بنفس القيمة الاخيرة.

و هذا مثال آخر للفهم أكثر :

```
int increment();

int main(int argc, char *argv[])
{
        printf("%d\n", increment());
        printf("%d\n", increment());
        printf("%d\n", increment());
        printf("%d\n", increment());

        return 0;
}

int increment()
{
        static int number= 0;

        number++;
        return number;
}
```

```
1
2
3
4
```

هنا، في المرة الأولى التي نطلب فيها الدالة `increment`، يتم إنشاء المتغير `number`. ثم نقوم بزيادة 1 إلى قيمته. و ما إن تنتهي الدالة لا يمسح المتغير.

عندما نطلب الدالة للمرة الثانية، يتم ببساطة قفز السطر الخاص بالتصريح بالمتغير، و لا نقوم بإعادة إنشاء المتغير بل فقط نعيد استعمال المتغير الذي أنشأناه سابقا. عندما يأخذ المتغير القيمة 1، تصبح قيمته 2 ثم 3 ثم 4 ... إلخ.

هذا النوع من المتغيرات ليس مستعملا بكثرة، لكن يمكنه مساعدتك في بعض الأحيان و لهذا ذكرته في هذا الكتاب.

الدوال المحلية لملف

لإنهاء حديثنا عن المتغيرات و الدوال، نتكلم قليلا حول نطاق الدوال. من المفروض، عندما تنشئ دالة، و التي هي شاملة بالنسبة لكل البرنامج، يمكن الوصول إليها من أي ملف `.c`.

قد تحتاج أحياناً إلى إنشاء دوال يمكن الوصول إليها فقط في الملفات التي تتواجد بها، و للقيام بذلك، أضف الكلمة المفتاحيّة `static` قبل الدالة كالتالي :

```c
static int triple(int number)
{
        // Instructions
}
```

و لا تنس النموذج أيضا :

```c
static int triple(int number);
```

إنتهى ! دالتك الساكنة `triple` لا يمكن استدعاؤها إلا من داخل دوال تنتمي لنفس الملف (مثلا `main.c`). إذا حاولت استدعاء الدالة `triple` من خلال دالة أخرى من ملف آخر (مثلا `display.c`)، لن يشتغل البرنامج لأن `triple` وقتها لن تكون مرئية.

لنلخص كلّ شيء يتعلّق بنطاق المتغيرات :

- المتغير الذي يتم التصريح عنه داخل دالة يتم حذفه في نهاية الدالة مباشرة . لا يمكن أن يكون مرئيا إلا داخل هذه الدالة.
- المتغير الذي يتم التصريح عنه في داخل دالة بالكلمة المفتاحية `static`، لا يحذف في نهاية الدالة لكنه يحتفظ بقيمته ما دام البرنامج يعمل.
- المتغير الذي يتم التصريح عنه خارج الدوال يسمى متغيرا شاملا، يكون مرئيا في كلّ الدوال و في كلّ ملفات المشروع.
- المتغير الشامل المصرّح عنه بالكلمة المفتاحية `static` مرئي فقط في الملف الذّي يتواجد به، و لا يكون مرئيا في دوال باقي الملفات.

و بالمثل، هذه هي النطاقات الممكنة للدوال :

- الدالة الّتي يتم التصريح عنها عادياً تكون مرئية في كل ملفات المشروع، يمكننا طلبها إذا من أي ملف كان.
- إذا أردنا أن تكون الدالة مرئية فقط في الملف الذي تتواجد به فيجب إضافة الكلمة المفتاحيّة `static` قبلها.

ملخّص

- البرنامج يحتوي العديد من الملفات `.c` . كقاعدة عامة، لكل ملف `.c` ملف مرافق له يحمل الإمتداد `.h` (الذي يعني ملفّا رأسيّا (**Header**)). الـ`.c` يحوي الدوال بينما `.h` يحوي النماذج (**Prototypes**)، أي توقيعات هذه الدوال.
  - يتم تضمين محتوى الملفات `.h` في أعلى الملفات `.c` بالاستعانة ببرنامج يسمّى **المعالج القبلي**.
  - يتم تحويل الملفات `.c` إلى ملفات ثنائية `.o` من طرف **المترجم**.
  - يتم تجميع الملفات `.o` في ملف تنفيذي (`.exe`) من طرف **محرر الروابط** (**Linker**).
- المتغير الذّي يتم التصريح عنه داخل دالة غير قابل للوصول إلا من داخل هذه الدالة. نتكلّم هنا عن **نطاق المتغيرات**.

الفصل 11

المؤشّرات (Pointers)

لقد حان الوقت لنكتشف المؤشرات. خذ نفسا عميقا قبل أن تقرر قراءة هذا الفصل لأنه لن يكون فصلاً للهو و المرح. تمثل المؤشرات واحداً من المبادئ الأكثر أهمية و حساسية في لغة الـC . و إن كنت أصرّ على أهميتها فهذا لأنه لا يمكنك البرمجة بـC دون معرفتها و فهمها جيّدا. المؤشرات موجودة في كلّ مكان، و لقد استعملتها من قبل دون أن تعلم بذلك.

كثير من المتعلّمين يصلون إلى المؤشرات و يواجهون صعوبات في فهمها. سنعمل على ألا يكون الأمر مماثلا بالنسبة لك. ضاعف التركيز و خذ الوقت اللازم لفهم المخططات التي سأقدمها لك في هذا الفصل.

1.11 مشكل مضجر بالفعل

واحد من أكبر المشاكل مع المؤشرات هي أنّه بالإضافة إلى أنها صعبة الاستيعاب قليلا بالنسبة للمبتدئين، فإن المتعلّم لا يعرف ما هي أهميتها و فيما يمكننا استعمالها.

يمكنني أن أقول لك بأن "المؤشرات لا يمكن الاستغناء عنها في أي برنامج C ، صدقني !"، لكنّي أعرف أن هذه الحجّة ليست كافية لك.

سأطرح عليك مشكلاً لا يمكنك حلّه إلا باستخدام المؤشرات. سيكون هذا الخيط الأحمر في هذا الفصل. سنعود إليه في نهاية هذا الفصل و سترى حلّه باستعمال ما تعلّمته في هذا الفصل.

إليك المشكل: أريد كتابة دالة تقوم بإرجاع قيمتين مختلفتين. ستجيبني "هذا مستحيل !". بالفعل، الدالة لا يمكنها إرجاع سوى قيمة واحدة.

```
int function()
{
	return value;
}
```

إذا استخدمنا `int` ترجع لنا قيمة من نوع `int` (بفضل التعليمة `return`).

يمكننا أيضا كتابة دالة لا تُرجع أية قيمة باستخدام الكلمة المفتاحية `void`.

```
void function()
{

}
```

لكن إرجاع قيمتين مختلفتين في نفس الوقت ... هذا أمر مستحيل لأنه لا يمكننا استعمال تعليمتي `return`.

لنفرض أنني أريد كتابة دالة أعطيها كمدخل عددا من الدقائق. تقوم الدالة بإرجاع عدد الساعات و الدقائق الموافقة لها.

- إذا أعطيت القيمة 45 الدالة ترجع 0 ساعة و 45 دقيقة.
- إذا أعطيت القيمة 60 الدالة ترجع القيمة 1 ساعة و 0 دقائق.
- إذا أعطيت القيمة 90 الدالة ترجع القيمة 1 ساعة و 30 دقيقة.

لنكن مجانين و لنجرّب ذلك :

```
#include <stdio.h>
#include <stdlib.h>

/* I put the prototype at the top.
Because it's a short code, I don't put it in a .h file.
In a real program I would have put the prototype
in a separate .h file of course */

void minutesDevision(int hours, int minutes);

int main(int argc, char *argv[])
{
        int hours = 0, minutes = 90;

/* We have a variable "minutes" equals to 90.
   after calling the function, I want from the variable"
hours" to take the value 1 and from my variable
"minutes" to take the value 30 */

        minutesDivision(hours, minutes);
        printf("%d hours and %d minutes", hours, minutes);
        return 0;
}

void minutesDevision(int hours, int minutes)
{
        hours = minutes / 60;  // 90 / 60 = 1
        minutes = minutes % 60; // 90 % 60 = 30
}
```

النتيجة :

```
0 hours and 90 minutes
```

لم تشتغل ! ما الأمر يا ترى ؟ في الواقع، عندما نبعث متغيرا إلى دالة، يتم إنشاء نسخة من المتغير، و لهذا فالمتغير `hours` في الدالة `minutesDevision` هو ليس نفسه الذي في الدالة `main` ! إنه فقط نسخة !

الدالة `minutesDevision` تقوم بعملها. ففي داخلها المتغيران `hours` و `minutes` يحملان القيمتين الصحيحتين : 1 و 30.

لكن بعد ذلك، تتوقف الدالة مباشرة عند الوصول إلى الحاضنة الغالقة، مثلما تعلمنا سابقا : المتغيرات الخاصة بدالة يتم حذفها مباشرة عند انتهاء الدالة. إذن النسخ عن المتغيرات `minutes` و `hours` تُمسح. نرجع بعد ذلك للدالة `main`، و التي فيها متغيرات `minutes` و `hours` تحملان القيمتين 0 و 90. لقد فشلنا !

> لاحظ إذن، بما أن الدالة تقوم بنسخ المتغيرات التي نعطيها لها، لست مضطراً لستمية متغيراتك بنفس الأسماء التي تحملها في الدالة الرئيسية `main`. و بالتالي يمكنك ببساطة كتابة :
> `void minutesDivision(int h, int m)`
> `h` للساعات و `m` للدقائق.
> إن كانت متغيراتك لم تُسمى بنفس الطريقة في الدالة و في `main` فهذا لا يطرح أيّ مشكل !

باختصار، يمكنك إعادة المشكل في كلّ الاتجاهات. يمكنك محاولة بعث قيمة باستخدام الدالة (باستخدام `return` و باستخدام النوع `int` للدالة) ، لكن لا يمكنك إعادة أكثر من قيمة واحدة من بين القيمتين، هذا مشكل مطروح إذن. كما لا يمكنك استعمال متغيرات شاملة لأن هذا أمر غير مستحسن إطلاقا.

حسناً المشكل لازال مطروحاً، كيف يمكننا حلّه باستخدام المؤشرات ؟

2.11 الذاكرة، مسألة عنوان

تذكير بالمكتسبات القبليّة

سأعود بك قليلاً إلى الوراء، هل تتذكر فصل المتغيرات ؟

أيّا كانت إجابتك، أنصحك بأن تعود إلى ذلك الفصل و تقرأ منه القسم الذي يحمل عنوان (مسألة ذاكرة). هناك مخطط مهم جداً سأقترحه عليك من جديد :

العنوان	القيمة
0	145
1	3.8028322
2	0.827551
3	3901930
...	...
3 448 765 900 126 (و بعض الأجزاء)	940.5118

هكذا نقوم تقريباً بتمثيل الذاكرة الحية (RAM) الخاصة بالحاسوب.

يجب قراءة المخطط سطراً بسطر، السطر الأول يمثل "الخانة" الخاصة بأول الذاكرة. لكل خانة رقم، هذا الرقم يمثل **عنوانها** (تذكّر هذا المصطلح جيدا). تحتوي الذاكرة على عدد كبير جداً من العناوين تبدأ من الرقم 0 و تنتهي بالرقم (ضع رقماً كبيراً جداً هنا). عدد العناوين التي تتوفر عليها تعتمد على حجم الذاكرة التي يحتوي عليها الجهاز الخاص بك.

في كلّ عنوان يمكننا تخزين عدد واحد فقط. لا يمكننا تخزين عددين في نفس العنوان.

الذاكرة ليست مصنوعة سوى لتخزين الأعداد. لا يمكنها تخزين لا حروف و لا جُمل. و للتخلص من هذا المشكل تم اختراع جدول يقوم بالربط بين الحروف و الأعداد. يقول الجدول مثلاً :"العدد 89 يمثّل الحرف Y". سنعود في فصل لاحق إلى كيفيّة معالجة الحروف. حاليّا، سنتكفي بالتكلّم عن عمل الذاكرة.

عنوان و قيمة

حينما تنشئ متغيراً `age` من نوع `int` مثلا، بكتابة :

```c
int age = 10;
```

يطلب البرنامج من نظام التشغيل (الويندوز مثلا) الإذن لاستعمال جزء من الذاكرة. نظام التشغيل يجيب بالإشارة إلى أي عنوان سيسمح لك بتخزين العدد.

هنا تكمن أحد أهم وظائف نظام التشغيل : نقول أنه يحجز الذاكرة للبرامج. يمكننا القول أنه هو القائد، يتحكم في كلّ برنامج و يتأكد من أن هذا الأخير له الإذن لاستعمال الذاكرة في المكان الذي يطلبه.

م

إن هذا هو السبب الرئيسي في توقف البرامج عن العمل : إذا حاول برنامجك الوصول إلى مكان غير مسموح له بالوصول إليه في الذاكرة، سيرفض نظام التشغيل و يوقف تشغيله بشكل عنيف (لأنه القائد هنا). بينما يتلقّى المستعمل نافذة خطأ تحتوي على رسالة تشير بأن البرنامج يحاول القيام بعملية غير لائقة.

لنعد للمتغير `age`. تم تخزين القيمة 10 في مكان ما من الذاكرة، لنقل مثلاً في العنوان رقم 4655. ما يحدث (و هذا دور المترجم) هو أن الكلمة `age` يتم تعويضها بالعنوان 4655 لحظة التنفيذ. مما يعني أنه في كلّ مرة قمت فيها بكتابة الكلمة `age` في الشفرة المصدرية، يتم تعوضيها بـ4655، و بهذا يرى الجهاز إلى أي عنوان في الذاكرة عليه الذهاب. و منه يجيب بكلّ فخر بأن المتغير `age` يحتوي القيمة 10.

نحن نعرف إذا كيف نسترجع قيمة متغير، يكفي بكلّ بساطة أن نكتب الكلمة `age` في الشفرة المصدرية. إذا أردنا إظهار السنّ، يمكننا استعمال الدالة `printf` :

```
printf("The value of variable age is : %d", age);
```

النتيجة على الشاشة :

```
The value of variable age is : 10
```

لا شيء جديد لحدّ الآن.

الخبر المثير لليوم

أنت تعرف كيف تظهر قيمة متغير، لكن هل تعرف أنه بإمكاننا أيضا إظهار عنوانه ؟

لكي نُظهر عنوان متغير، نستعمل الإشارة `%p` (الحرف p مأخوذ من الكلمة Pointer) في الدالة `printf`. أي أننا لن نبعث للدالة `printf` المتغير في حدّ ذاته لكن نبعث لها عنوانه. و لفعل هذا، يجب عليك استعمال الإشارة `&` أمام المتغير `age` كما طلبت منك أن تفعل مع الدالة `scanf` من قبل دون أن أشرح لك لماذا.

أكتب إذا:

```
printf("The address of the variable age is  : %p", &age);
```

النتيجة :

```
The address of the variable age is : 0023FF74
```

ما تراه هنا هو عنوان المتغير `age` في اللحظة التي طلبتُ فيها تنفيذ البرنامج من طرف حاسوبي. نعم نعم، 0023FF74 هو رقم، هو فقط مكتوب في النظام الست عشري (Hexadecimal) عوض النظام العشري الذي تعوّدنا عليه. لو تقوم بتعويض `%p` بـ `%d` فإنه سيظهر لك رقما عشريا كما تعوّدنا.

م

إذا شغلّت البرنامج على حاسوبك فمن المؤكّد أن تحصل على عنوان آخر. الأمر يعتمد على المكان في الذاكرة، البرامج المشتغلة، إلخ. فيستحيل أن نتوقع العنوان الذّي سيتم تخزين المتغيّر فيه. إذا قمت بتشغيل البرنامج عدة مرات الواحدة تلو الأخرى قد تحصل على نفس النتيجة كون الذاكرة لم تتغير في ذلك الزمن القصير. لكن بالمقابل إن أعدت تشغيل الحاسوب فستتحصل بكل تأكيد على نتائج مختلفة.

إلى أين أريد الوصول بكلّ هذا ؟ أريدك أن تتذكّر التالي :

- `age` : تعني قيمة المتغير.
- `&age` : تعني عنوان المتغير.

عند استخدام `age` سيقرأ الحاسوب قيمة المتغيّر في الذاكرة. أمّا عند استخدام `&age` فسيعيد العنوان الذي يوجد فيه المتغيّر.

3.11 استعمال المؤشرات

لحدّ الآن، قمنا فقط بإنشاء متغيرات تحتوي على أعداد. الآن سنتعلّم كيف ننشئ متغيرات تحتوي على عناوين : هذا ما نسميه بالمؤشرات.

؟

لكن ... العناوين هي أعداد أيضاً، أليس كذلك ؟ هذا يعني أننا سنخزن أعداداً دائما !

هذا صحيح، لكن لهذه الأعداد معنى آخر : هي تشير إلى عنوان متغير آخر في الذاكرة.

إنشاء مؤشّر

لإنشاء متغير من نوع مؤشّر، يجب علينا أن نضيف الرمز `*` أمام إسم المتغير :

```
1 int *myPointer;
```

م

لاحظ أنه يمكننا أيضا أن نكتب `int* myPointer;` لهذا نفس المعنى. لكن الطريقة الأولى هي المفضّلة. في الواقع، إن كنت تريد التصريح عن العديد من المؤشرات في نفس السطر، سيكون عليك أن تعيد كتابة النجمة أمام كل اسم : `int *pointer1, *pointer2, *pointer3;`

كما قلت لك، من المهمّ أن تقوم بإعطاء قيم إبتدائية للمتغيرات منذ البداية، و ذلك بإعطائها القيمة 0 مثلا ! إنه من المهم أكثر أن تفعل نفس الشيء مع المؤشرات.
لتهيئة مؤشّر، نعطيه قيمة افتراضية، لا نستعمل غالبا القيمة 0 و لكن الكلمة المفتاحية `NULL` (أكتبها بأحرف كبيرة).

```
int *myPointer = NULL;
```

هنا لدينا مؤشر يحمل القيمة الإبتدائية `NULL`. هكذا ستعرف لاحقاً في البرنامج أن المؤشر لا يحتوي على أي عنوان.

ما الذي يحصل ؟ ستقوم هذه الشفرة المصدرية بحجز خانة في الذاكرة كما لو أننا أنشأنا متغيراً عادياً. الشيء الذي يتغير هو أن المؤشر سيحتوي عنوانا. عنوان متغير آخر.

لم لا عنوان المتغير `age` ؟ أنت تعرف الآن كيف تشير إلى عنوان متغير في مكان قيمته (باستعمال الرمز `&`)، هيا بنا إذن ! هذا ما عليك كتابته :

```
int age = 10;
int *PointerOnAge = &age;
```

السطر الأول يعني : "أنشئ متغيرا من نوع `int` يحمل القيمة 10". السطر الثاني يعني "أنشئ متغيراً من نوع مؤشّر قيمته هي عنوان المتغير `age`".

يقوم إذا السطر الثاني بمهمّتين معاً. لكي لا تختلط عليك الأمور، إعلم أنه يمكننا تقسيم السطر إلى سطرين :

```
int age = 10;
int *PointerOnAge; // 1) Means "I create the pointer"
PointerOnAge = &age; // 2) Means the pointer "PointerOnAge contains the address
    of age"
```

يمكنك الملاحظة أنه لا يوجد في لغة الـC نوع نسميه `pointer` كالنوع `int` و `double`. أي أنه لا يمكننا أن نكتب :

```
pointer PointerOnAge;
```

في مكان هذا، نستعمل الرمز `*`، و لكن نستمر في كتابة `int`. ماذا يعني هذا ؟ في الواقع يجب أن نشير إلى نوع المتغير الذي سيحوي عنوانه المؤشر. بما أن المؤشر `PointerOnAge` سيحتوي عنوان المتغير `age` (الذي هو من نوع `int`)، إذا فالمؤشر يجب أن يكون من نوع `int*` ! إذا كان المتغير من نوع `double` فإنه يجب عليّ أن أكتب `double *myPointer`.

إصطلاح : نقول بأن المؤشّر `PointerOnAge` يؤشّر على المتغير `age`.

المخطط التالي يلخّص ما يحصل في الذاكرة :

	العنوان	القيمة
	0	145
	1	3
	2	82
PointerOnAge	3	177450
	...	...
age	177450	10
	...	...
	3 448 765 900 126 (و بعض الأجزاء)	940

في هذا المخطط، تم تعويض المتغير `age` بالعنوان 177450 (أنت ترى بأن قيمته هي 10)، و المؤشر `PointerOnAge` تم تعويضه بالعنوان 3 (هذه محض صدفة).

حينما يتم إنشاء المؤشر، يقوم نظام التشغيل بحجز خانة في الذاكرة كما فعل مع المتغير `age`. الشيء المختلف هنا هو أن المتغير `PointerOnAge` له معنى آخر. أنظر للمخطط جيداً : قيمته هي عنوان المتغير `age`.

هذا، عزيزي القارئ، هو السرّ المطلق من وراء كتابة البرامج في لغة الـC. بهذا نحن ندخل في عالم المؤشرات العجيب !

؟

و ... ما هي فائدة هذا ؟

هذا لا يقوم بتحويل الحاسوب إلى آلة صنع القهوة، طبعا. لكن الآن لدينا المؤشر `PointerOnAge` يحتوي عنوان المتغير `age`.

فلنحاول رؤية ما يحتويه المؤشر بالإستعانة بالدالة `printf` :

```c
int age = 10;
int *PointerOnAge = &age;
printf("%d", PointerOnAge);
```

```
177450
```

هذا ليس مفاجئاً، نحن نطلب قيمة `PointerOnAge` و قيمته هي عنوان المتغير `age` (أي 177450).

ماذا نفعل لكي نطلب قيمة المتغير المتواجدة في العنوان الذي يشير إليه المؤشّر `PointerOnAge` ؟ يجب أن نضع الرمز `*` أمام إسم المؤشّر :

```
1 int age = 10;
2 int *PointerOnAge = &age;
3 printf("%d, *PointerOnAge);
```

```
10
```

ها قد وصلنا ! بوضع الرمز `*` أمام إسم المؤشّر، يمكننا الوصول إلى قيمة المتغير `age`.

لو استعملنا الرمز `&` أمام اسم المؤشّر، سنتحصل على العنوان الذي يتواجد به المؤشّر (هنا الرقم 3).

> ؟ ماذا نربح هنا ؟ لقد نجحنا في تعقيد الأمور لا أكثر. لم نكن نحتاج إلى مؤشّر لنظهر قيمة المتغير `age` !

هذا السؤال (الذي لا مفر من طرحه) شرعي، حاليّا الهدف ليس واضحا، لكن شيئاً فشيئاً، و مع تقدّم الفصول، ستفهم بأن كلّ هذه المبادئ لم يتم اختراعها من أجل تعقيد الأمور بكلّ سذاجة.

المهم هو أن تفهم المبدأ الآن و بعده ستتوضح الأمور لوحدها رويداً رويداً.

ما يجب تذكّره

هذا ما يفترض أن تكون قد فهمته و يجب عليك تذكره لباقي الفصل :

- بالنسبة لمتغير كالمتغير `age` :
  - `age` تعني: "أريد قيمة المتغير `age`"
  - `&age` تعني: "أريد العنوان الذي يتواجد به المتغير `age`"
- بالنسبة لمؤشّر كالمؤشّر `PointerOnAge` :
  - `PointerOnAge` تعني: "أريد قيمة `PointerOnAge`" (هذه القيمة هي عنوان).
  - `*PointerOnAge` تعني : "أريد قيمة المتغير المتواجد في العنوان الذي يحتويه المؤشر `PointerOnAge`".

اكتفِ بفهم و حفظ النقاط الأربع السابقة، قم باختبارات و تأكد أنها تعمل. المخطط التالي سيعينك على فهم هذه النقاط :

العنوان	القيمة
0	145
1	3
2	82
3	177450 *PointerOnAge*
...	...
177450 &age	10 age *PointerOnAge*
...	...
3 448 765 900 126 (و بعض الأجزاء)	940

احذر في عدم الخلط بين مختلف تفسيرات النجمة ! حينما تصرّح عن مؤشّر، فإنّ النجمة تشير إلى أننا بصدد إنشاء مؤشر : `int *PointerOnAge;` بينما لما نكتبها بجانب اسم المؤشر بكتابة : `printf("%d", *PointerOnAge);` فهذا يعني : أريد قيمة المتغير التي يشير إليها المؤشر `PointerOnAge`.

كل هذه المعلومات مبدئية. يجب أن تعرفها و الأهمّ أن تتعلّمها عن ظهر قلب. لا تتردّد في قراءة و إعادة قراءة ما قدّمته لك. لا يمكنني أن أعاتبك إذا لم تفهم من المرّة الأولى، و هذا ليس أمرا مخجلا. عادة ما تحتاج لعدّة أيّام لكي تفهمها جيّدا و بضعة شهور لتفهم كلّ شيء.

إذا كنت تشعر بأنك ضائع قليلاً، فكر في الأشخاص المحترفين في البرمجة : لا أحد منهم فهم مبدأ عمل المؤشرات من المرّة الأولى. و إن كان هذا الشخص موجوداً، أطلب منك أن تقدّمه لي الآن.

4.11 إرسال مؤشّر إلى دالة

أهم فائدة للمؤشرات (لكنها ليست الوحيدة) هي أنه بامكاننا إرسالها إلى دالة كي تقوم بتعديل مباشر على قيمة متغير في الذاكرة لا لنسخة منها كما رأينا سابقاً.

كيف يعمل هذا ؟ هناك الكثير من الطرق لفعل هذا. إليك مثالاً :

```c
void triplePointer(int *numberPointer);

int main(int argc, char *argv[])
{
	int number = 5;

	triplePointer(&number); // We send the address of number to the function
	printf("%d", number); // We display the variable number. The function has directly modified its value because it knows the address of the variable

	return 0;
}

void triplePointeur(int *numberPointer)
{
	*numberPointer *= 3; // We multiply by 3 the value of number
}
```

```
15
```

الدالة `triplePointer` تأخذ معاملا من نوع `int*` (أي مؤشّر على `int`).
إليك ما يحدث بالترتيب، إنطلاقاً من بداية الدالة `main` :

1. يتم إنشاء متغير `number` في الدالة `main`، نُسنِد له القيمة 5، أنت تعرف هذا.

2. نستدعي الدالة `triplePointer` و نرسل لها كمعامل عنوان المتغير `number`.

3. تتلقّى الدالة `triplePointer` المتغير في المؤشر `numberPointer`. داخل الدالة `triplePointer`، لدينا إذن المؤشر `numberPointer` الذي يؤشر على المتغير `number`.

4. و الآن بما أنه لدينا مؤشر على المتغير `number`، يمكننا تغيير قيمة المتغير `number` مباشرة في الذاكرة ! يكفي استعمال `*numberPointer` للإشارة إلى المتغير `number` ! كمثال، نقوم باختبار بسيط : نضرب المتغير في 3.

5. بالعودة إلى `main`، يحتوي المتغير `number` القيمة 15 لأن الدالة `triplePointer` قامت بتعديل قيمته مباشرة.

بالطبع، كان بإمكاني استعمال `return` بسيط كما تعلمنا القيام بذلك في الفصل الخاص بالدوال. لكن الهدف هنا، هو أنه بهذه الطريقة، أي باستعمال المؤشرات، يمكننا تعديل قيم الكثير من المتغيرات في الذاكرة (يمكننا إذا "إرجاع" الكثير من القيم !). لسنا محدودين بقيمة واحدة بعد الآن !

> ؟ ما الفائدة الآن من استعمال `return` في دالة إذا كان بإمكاننا استعمال المؤشرات لتعديل قيم المتغيرات ؟

هذا يعتمد عليك و على برنامجك. القرار لك. يجب أن تعرف أن تعليمة `return` مستعملة بكثرة في لغة الـC. غالباً نستعملها لإرجاع شفرة الخطأ : الدالة ترجع القيمة 1 (صحيح) إذا اشتغل كل شيء على ما يُرام، و 0 (خطأ) إذا حدث خطأ ما أثناء تشغيل البرنامج.

طريقة أخرى لإرسال مؤشر إلى دالة

في الشفرة المصدرية التي رأيناها، لم يكن هناك مؤشر في الدالة `main` ، و إنما فقط متغير `number` . المؤشر الوحيد كان في الدالة `triplePointer` (من نوع `int*`).

يجب أن تعرف أنه هناك طريقة أخرى لكتابة الشفرة المصدرية السابقة، وذلك بإضافة مؤشر في الدالة `main` :

```
void triplePointer(int *numberPointer);

int main(int argc, char *argv[])
{
	int number = 5;
	int *pointer = &number; // pointer gets the address of number
	triplePointer(pointer); // We send pointer (the address of number) to the function
	printf("%d", *pointeur);
	return 0;
}

void triplePointer(int *numberPointer)
{
	*numberPointer *= 3; // We multiply by 3 the value of number
}
```

قارن بين الشفرتين المصدريتين السابقتين، ستجد بأن هناك اختلافا بينهما، لكن النتيجة واحدة :

```
15
```

ما يهم، هو بعث عنوان المتغير `number` إلى الدالة. لكن المؤشر يحمل عنوان المتغير `number` ، فهذا جيد من هذه الناحية ! نحن فقط نقوم بالأمر بطريقة مختلفة بإنشاء المؤشر في الدالة `main` .

في الدالة `printf` (فقط من أجل التمرين)، أُظهر محتوى المتغير `number` بكتابة `*pointer` . لاحظ أنه في مكان هذا، يمكنني كتابة `number` : ستكون النتيجة هي نفسها لأنّ `*pointer` و `number` يشيران إلى نفس الخانة بالذاكرة.

في البرنامج "أكثر أو أقل"، استعملنا مؤشراً دون أن نعرف بذلك. كان ذلك عند استدعاء الدالة `scanf` . بالفعل، دور هذه الدالة هو قراءة ما كتبه المستعمل في لوحة المفاتيح و إرجاع النتيجة. لكي تتمكن الدالة من تغيير محتوى المتغير مباشرة أي تضع بها الكلمة أو الجملة التي تمت كتابتها في لوحة المفاتيح، احتاجت لعنوان المتغير :

```
int number = 0;
scanf("%d", &number);
```

تعمل الدالة بمؤشر على المتغير `number` و بهذا يمكنها تغيير قيمته مباشرة.
كما رأينا الآن، يمكننا إنشاء مؤشر و نبعثه للدالة `scanf` :

```
int number = 0;
int *pointer = &number;
scanf("%d", pointer);
```

إنتبه كي لا تضع الرمز `&` امام اسم مؤشر في الدالة `scanf` ! هنا `pointer` يحتوي هو نفسه عنوان المتغير `number`، لذا أنت لا تحتاج لوضع `&` ! إذا فعلت هذا فإنك ستبعث العنوان الذي يتواجد به المؤشر : و لكننا بحاجة لعنوان `number` !

5.11 من الذي قال "مشكل مضجر بالفعل" ؟

اقتربنا من نهاية الفصل، حان الوقت لإيجاد الخيط الأحمر. إذا كنت قد فهمت الفصل، فأنت قادر على حلّ المشكل الآن، ما الذي تقوله. حاول حل المشكل و إليك الحل للمقارنة :

```
void minutesDivision(int* hoursPointer, int* minutesPointer);

int main(int argc, char *argv[])
{
        int hours = 0, minutes = 90;
        // We send the addresses of hours and minutes
        minutesDivision(&hours, &minutes);
        // This time, the values are modified !
        printf("%d hours and %d minutes", hours, minutes);
        return 0;
}

void minutesDivision(int* hoursPointer, int* minutesPointer)
{
        /* Don't forget to put a star next to the pointer's name ! so you can
           modify the value of the variable and not its address ! You don't
           want to divide addresses, do you ? */
        *hoursPointer = *minutesPointer / 60;
        *minutesPointer = *minutesPointer % 60;
}
```

النتيجة :

```
1 hours and 30 minutes
```

لا شيء يجب أن يفاجئك في هذه الشفرة المصدرية. كما أفعل في كلّ مرة، سأشرح ما يحصل هنا خطوة بخطوة لأتأكد من أن جميع القرّاء قد فهموا المبدأ. إنه فصل مهم و لهذا عليك أن تبذل جهداً كبيراً لتفهم كما أفعل أنا من أجلك !

- يتم إنشاء المتغيرين `hours` و `minutes` في الدالة `main`.

- نرسل للدالة `minutesDevision` عنواني المتغيرين `hours` و `minutes`.
- تقوم الدالة `minutesDevision` باسترجاع قيمتي المتغيرين في مؤشرين يدعيان `hoursPointer` و `minutesPointer`. لاحظ أنه لا يهم الاسم هنا أيضاً. كان بإمكاني تسميتهما `m` و `h`، أو حتى `hours` و `minutes` لم أقم بهذا لكي لا تخلط بين المتغيرين الخاصين بالدالة الرئيسية مع هذين المؤشرين اللذان يختلفان معهما.
- تقوم الدالة `minutesDivision` بتعديل مباشر لقيمتي المتغيرين `hours` و `minutes` في الذاكرة لأنها تملك عنوانيهما في مؤشرين. الشيء الوحيد الذي يمكنه أن يشكل مشكلا و يجب أن أبقيه ببالي هو وضع النجمة أمام اسمي المؤشرين إذا أردت أن أغير قيمتي المتغيرين `hours` و `minutes`. إذا لم نفعل هذا، فإننا سنغير العناوين الموجودين في المؤشرين، و هذا لا ينفع في شيء.

> م
>
> كثير من القرّاء سيشيرون إلى أنه يمكن حل المشكل دون اللجوء إلى المؤشرات. نعم هذا صحيح، لكنّ هذا سيضطرنا لتجاوز القواعد التي وضعناها معاً. يمكننا استعمال المتغيرات الشاملة (و كما قلتُ هذا شيء سيء)، و يمكننا أيضاً وضع الدالة `printf` داخل الدالة `minutesDivision` (لكننا نحن نريد وضع العرض في الدالة الرئيسية). هذا يعني أنه يمكننا دائما حل المشكل باعتماد طرق غير لائقة مما يجعلك تشك في الفائدة من المؤشرات. لكن تأكد من أنك ستجد بأن استعمال المؤشرات أمر بديهي مع التقدم في الكتاب.

ملخّص

- كل متغير يتم تخزينه في **عنوان** معين من الذاكرة.
- تشبه **المؤشرات** المتغيرات لكنها مخصصة لتخزين العناوين التي تتواجد بها المتغيرات في الذاكرة.
- إذا وضعنا الرمز `&` أمام اسم متغير فإننا نحصل على العنوان الذي يتواجد به المتغير، مثلا : `&age`.
- إذا وضعنا الرمز `*` امام إسم مؤشر نحصل على قيمة المتغير التي يؤشّر عليها المؤشّر.
- تعتبر المؤشرات إحدى المبادئ الأساسية في لغة الـC، تبدو صعبة في البداية لكن عليك أن تفهمها لأن الكثير من المبادئ الأخرى تقوم عليها.

الفصل 12

الجداول (Arrays)

هذا الفصل هو ملحق مباشر للفصل المتعلق بالمؤشرات، و سيعلّمك أهميتها أكثر. إن كنت تعتقد بأنك قادر على تفادي المؤشرات فأنت مخطئ ! هي في كلّ مكان في لغة الـC. لقد حذّرتك !

سنتعلم في هذا الفصل كيف ننشئ متغيرات من نوع "جداول". الجدوال مهمّة للغاية في لغة الـC لأنها تساعد في تنظيم سلسلة من القيم.

نبدأ هذا الفصل ببعض الشروحات و التفسيرات حول كيفية عمل الجداول في الذاكرة (سأقدم لك الكثير من المخططات التفسيرية). هذه المقدمات حول الذاكرة مهمة جداً : ستساعدك في في معرفة عمل الجداول. فمن المستحسن أن يعرف المبرمج ما يقوم به كي يتحكم في برامجه أكثر، أليس كذلك ؟

1.12 الجداول في الذاكرة

"الجداول هي تتابع متغيرات من نفس النوع، موجودة في مكان متواصل من الذاكرة."

أعرف أن هذا التعريف يشبه قليلا تعريف القاموس. لهذا فسأوضح بطريقة أخرى، فعلياً، الجدول عبارة عن "متغيّرات ضخمة" يمكن لها أن تحتوي على أعداد كبيرة من نفس النوع (`char`، `long`، `int`، `double`...).

للجدول طول محدد. يمكنه أن يكون 2، 3، 10 خانات، 150، 2500 خانة، أنت من يحدد العدد. المخطط التالي مثال عن جدول يحجز 4 خانات بدءاً بالعنوان 1600 :

القيمة	العنوان
10	1600
23	1601
505	1602
8	1603

عندما تطلب إنشاء جدول يحجز 4 خانات في الذاكرة، سيطلب برنامجك من نظام التشغيل أن يسمح له باستغلال 4 خانات في الذاكرة، و يجب ان تكون هذه الخانات متتالية يعني الواحدة بجانب الأخرى. و كما ترى أعلاه فالخانات متتابعة 1600، 1601، 1602، 1603 فلا يوجد "فراغ" بينها.

أخيراً، كل خانة تحتوي عددا من نفس النوع. فإن كان الجدول من نوع `int` فإن كلّ خانة يجب أن تحتوي عددا من نوع `int`. و بهذا نفهم أنه لا يمكننا وضع نوع `int` مع `double` في الجدول نفسه.

و كتلخيص، هذا أهم ما يجب أن تعرفه بخصوص الجداول :

- عندما يتم إنشاء جدول، يأخذ مكانا متواصلاً في الذاكرة. بحيث تكون الخانات متجاورة الواحدة تلو الأخرى.
- كل خانات الجدول تكون من نفس النوع، فجدول الـ `int` يمكن أن يحمل فقط `int`، و لا أي نوع آخر.

2.12 تعريف جدول

كي نبدأ سننشئ جدولا من 4 أعداد من نوع `int` :

```
int table[4];
```

هذا كلّ شيء. يكفي إذن أن تضيف قوسين مربعين (`[` و `]`) عدد الخانات التي تريد أن يحجزها جدولك، و اعلم أنه لا يوجد حدود (إلا إن تجاوزت الحدّ الذي تسمح به ذاكرة جهازك طبعا).

و لكن الآن، كيف نصل لخانة ما في الجدول ؟ هذا سهل، تكفي كتابة `table[cellNumber]`.

X

احذر : كل جدول يجب أن يبدأ بالفهرس (Index) رقم 0 ! جدولنا متكوّن من 4 `int` إذن فالفهارس المتوفرة هي : 0، 1، 2 و 3. لا وجود للفهرس 4 في جدول من 4 خانات ! هذا مصدر أخطاء متداولة، فلا تغفل عنه !

إذا كنت أريد أن أضع في جدولي نفس القيم التي في المخطط فيجب إذا أن أكتُب :

```
int table[4];
table[0] = 10;
table[1] = 23;
table[2] = 505;
table[3] = 8;
```

؟ لازلت لا أرى العلاقة بين المؤشرات و الجداول ؟

في الواقع، لو تكتب فقط `table` فستحصل على مؤشر، و هو مؤشر على الخانة الأولى من الجدول، قم باختبار التالي :

```
int table[4];
printf("%d", table);
```

النتيجة ستظهر لك العنوان الذي يتواجد به `table` :

```
1600
```

بينما إذا قمت بوضع فهرس الخانة بين قوسين مربعين، فستحصل على القيمة :

```
int table[4];
printf("%d", table[0]);
```

```
10
```

نفس الشيء بالنسبة للفهارس الأخرى. بما أن `table` هو مؤشر، يمكننا استعمال الرمز `*` للحصول على القيمة الأولى :

```
int table[4];
printf("%d", *table);
```

```
10
```

يمكن أيضا الحصول على قيمة الخانة الثانية بكتابة `*(table + 1)` (أي عنوان الجدول + 1). لذا فهذان السطران متماثلان :

```
table[1] // Returns the value of the second cell (the first is 0)
*(table + 1) // Same thing : returns the value of the second cell.
```

لذا فعند كتابة `table[0]` ، فأنت تطلب قيمة الخانة التي تتواجد بعنوان الجدول + 0 خانة، أي 1600.
و إذا كتبت `table[1]` فإنك تطلب القيمة المتواجدة في عنوان الجدول + 1 خانة، أي 1601.
و هكذا من أجل الباقي.

الجداول ذات الحجم المتغيّر

هناك عدة نسخ من لغة C.
نسخة حديثة منها تدعى C99 تسمح بإنشاء جداول ذات حجم متغيّر. يعني أن حجم الجداول يمكن أن يكون معرّفا بمتغير.

```
int size = 5;
int table[size];
```

إلا أن هذه الكتابة ليست مفهومة بالنسبة لكل المترجمات (Compilers) فبعضها تتوقف في السطر الثاني. إن لغة الC التّي اعلمك إياها منذ البداية (تدعى C89) لا تسمح بهذا النوع من الكتابات. و لذا يمكننا القول أن فعل هذه الأشياء أمر ممنوع.

يجب أن نتفق على شيء و هو : لا تملك الحق في وضع متغير بين القوسين المربعين من أجل تعريف حجم الجدول. حتى و إن كان المتغير ثابتا ! يجب على طول الجدول أن يأخذ قيمة ثابتة، و لهذا عليك أن تحدده كعدد :

```
int table[5];
```

> ؟ إذن ... هل من الممنوع إنشاء جدول يعتمد حجمه على قيمة متغير ؟

بلى إنه ممكن حتى مع C89. لكن لفعل هذا سنعتمد على تقنية أخرى (أكيدة أكثر و تعمل مع كل المترجمات) تدعى بـالحجز الحي (Dynamic allocation). سندرسها في مرحلة متقدمة من هذا الكتاب.

3.12 تصفح جدول

لنفرض أنني أريد الآن أن أُظهر كل قيم خانات الجدول.
يمكنني أن أستدعي الدالة `printf` بالقدر الذي يحتويه الجدول من خانات. لكن سيكون الأمر ثقيلاً و مليئاً بالتكرار، و تخيل حجم الشفرة المصدرية لو أننا أردنا إظهار قيم الجدول واحدة بواحدة !

الأحسن هو أن نستعين بحلقة. لم لا حلقة `for` ؟ فهي الأنسب لتصفح الجداول :

```
int main(int argc, char *argv[])
{
	int table[4], i = 0;
	table[0] = 10;
	table[1] = 23;
	table[2] = 505;
	table[3] = 8;
	for (i = 0 ; i < 4 ; i++)
	{
		printf("%d\n", table[i]);
	}
	return 0;
}
```

```
10
23
505
8
```

إن حلقتنا تتصفح الجدول بمساعدة متغير يسمى `i` (اسم شائع لدى المبرجين يخص المتغير الذي يستخدم لتصفح جدول !).

إن الشيء العَمَلِيّ خاصة، هو أنه بإمكاننا وضع متغير داخل قوسين مربعين بالفعل. فالمتغير كان ممنوعا في مرحلة إنشاء الجدول (لتعريف حجمه). لكن و لحسن الحظ، فهو مسموح من أجل "تصفح" الجدول، أي إظهار قيمه !
هنا قد نعطي المتغير `i` بشكل متتالي القيم 0، 1، 2، 3. بهذا سنقوم إذن بإظهار قيمة `table[0]`، `table[1]`، `table[2]` و `table[3]` !

x

إنتبه لعدم محاولة إظهار قيمة `table[4]` ! لجدول من 4 خانات يتضمن الفهارس 0، 1، 2، 3 فقط. فإن حاولت عرض `table[4]` فإمّا أن تحصل على قيمة عشوائية، أو أن يظهر لك خطأ جميل، نظام التشغيل يوقف برنامجك فهو يحاول الوصول لعنوان لا ينتمي إليه.

تهيئة جدول

الآن و مادمنا قد عرفنا كيف نتصفح جدولا، يمكننا أن نضبط كل قيمه على 0 باستخدام حلقة !

إن القيام بتصفح جدول لضبط كلّ قيمه على الصفر أمر يمكن القيام به بمستواك هذا :

```
1  int main(int argc, char *argv[])
2  {
3          int table[4], i = 0;
4          // Initialization of the table
5          for (i = 0 ; i < 4 ; i++)
6          {
7                  table[i] = 0;
8          }
9          // Displaying the values of the table to check
10         for (i = 0 ; i < 4 ; i++)
11         {
12                 printf("%d\n", table[i]);
13         }
14         return 0;
15 }
```

```
0
0
0
0
```

طريقة أخرى للتهيئة

يجب أن تعرف أنه هناك طريقة أخرى لإعطاء قيم ابتدائية لجدول في لغة الـC.
و هذه الطريقة تعمل بكتابة `table[4] = {value1, value2, value3, value4}` . ببساطة، تضع القيم بين حاضنتين واحدة تلو الاخرى و تفصل بينها بفواصل.

```
int main(int argc, char *argv[])
{
        int table[4] = {0, 0, 0, 0}, i = 0;
        for (i = 0 ; i < 4 ; i++)
        {
        printf("%d\n", table[i]);
        }
        return 0;
}
```

```
0
0
0
0
```

و هناك أحسن من هذه الطريقة : بإمكانك تعريف القيم الخاصة بالخانات الأولى من الجدول و كل التي لم تُشر إليها ستُضبط على 0.

أي أنني إن قمت بكتابة التالي :

```
int table[4] = {10, 23}; // Inserted values : 10, 23, 0, 0
```

الخانة 0 تأخذ القيمة 10 و الخانة 1 تأخذ القيمة 23 و كل الخانات المتبقية تأخذ القيمة 0 (افتراضياً).

كيف نضبط كل الجدول على الصفر بمعرفة هذا ؟
يكفي أن تضبط القيمة الأولى على 0، و كل القيم الاخرى غير المشار إليها تأخذ القيمة 0.

```
int table[4] = {0}; // All the columns of the table will be initialised to 0
```

هذه التقنية لها شيء مميز و هو أنها تعمل مع أي جدول مهما كان حجمه (في هذا المثال نجحت الطريقة مع 4 خانات و ستنجح لو كان الجدول بـ100 خانة أيضا).

X

احذر، قد تصادفك الكتابة التالية : `int table[4] = {1};` و التي تعني إدخال القيم التالية في الجدول : 1، 0، 0، 0.
خلافاً لما يعتقده الكثير، لن يتم تهيئة كل خانات الجدول على 1. بل الخانة الأولى هي الوحيدة التي تضبط على 1 أما الباقي فعلى 0. لا يمكننا إذن القيام بتهيئة كل الجدول على القيمة 1 تلقائيّا، إلّا باستعمال حلقة.

4.12 تمرير جدول لدالة

في كثير من الأوقات ستحتاج لإظهار محتوى كلّ الجدول، فلما لا نقوم بكتابة دالة تقوم بهذا ؟ بهذا ستكتشف كيف نمرر جدولا إلى دالة (و هذا يساعدني).

يتطلب الأمر أن تبعث معلومتين للدالة. الجدول (أي عنوان الجدول) و أيضا حجمه خاصّة !
بالفعل، يجب أن تكون دالتنا قادرة على تهيئة جدول مهما كان حجمه. لكن في دالتنا، أنت لا تعرف حجم جدولك و لهذا يجب أن ترسل متغيرا يحمل الاسم `tableSize` مثلا.

و كما قلت لك مسبقا، يمكننا إعتبار `table` مؤشرا، و لهذا يمكننا أن نرسله للدالة مثلما نرسل مؤشرا عاديا.

```
1  // Prototype of the display function
2  void display(int *table, int tableSize);
3
4  int main(int argc, char *argv[])
5  {
6          int table[4] = {10, 15, 3};
7          // We display the content of the table
8          display(table, 4);
9          return 0;
10 }
11
12 void display(int *table, int tableSize)
13 {
14         int i;
15         for (i = 0 ; i < tableSize; i++)
16         {
17                 printf("%d\n", table[i]);
18         }
19 }
```

```
10
15
3
0
```

الدالة غير مختلفة عن التي درسناها في فصل المؤشرات، فهي تأخذ كمعامل مؤشراً نحو `int` (جدولنا) و أيضا حجمه (مهم جدا كي نعرف متى نتوقف الحلقة !).
كل محتوى الجدول تُظهره الدالة بواسطة الحلقة.

لاحظ أنّه توجد طريقة أخرى للإشارة إلى أن الدالة تستقبل جدولا. بدلا من أن نشير أن الدالة تستقبل `int *table`، أكتب التالي :

```
1  void display(int table[], int tableSize)
```

هذا يعني تماما نفس الشيء، و لكن وجود القوسين المربعين يُعلمان المبرج بأن الدالة تأخذ جدولا و ليس مؤشرا عاديا و هذا يزيل الغموض.

شخصيا أستعمل دائما القوسين المربعين في دوالي لكي أُظهر بأن الدالة تنتظر جدولا. أنصحك باستعمال نفس الطريقة. ليس مهما وضع حجم الجدول بين القوسين المربعين هذه المرة.

بعض التمارين !

لدي أفكار متعددة عن تمارين متعلقة بالجداول ستساعدك على التدريب ! و فكرة التمارين هي إنشاء دوال تعمل على الجداول.

و كتحدي، ستجد هنا نصوص التمارين فقط، و لن أعطي الإجابة كي أجبرك على الاجتهاد في إيجاد الحلول، فإن لم تستطع فيمكنك زيارة المنتديات لطرح أسئلتك.

التمرين 1

أنشئ دالة `tableSum` ترجع مجموع القيم الموجودة في الجدول (استعمل `return` لإرجاع النتيجة) و للمساعدة، هذا هو نموذج الدالة :

```
int tableSum(int table[], int tableSize);
```

التمرين 2

أنشئ دالة `tableAverage` تحسب و تُرجع معدّل القيم الموجودة في الجدول. تفضل النموذج :

```
double tableAverage(int table[], int tableSize);
```

الدالة ترجع `double` فالمعدّل عادة هو قيمة عشرية.

التمرين 3

أنشئ دالة `tableCopy` التي تأخذ كمعاملات جدولين، حيث تقوم بنسخ محتوى الجدول الأول في الجدول الثاني، تفضل النموذج :

```
void tableCopy(int originalTable[], int copyTable[], int tableSize);
```

التمرين 4

أنشئ دالّة `tableMaximum` دورها إسناد 0 لكلّ خانة جدول تحوي قيمة أكبر من القيمة العظمى. هذه الدالة تأخذ كمعاملات الجدول و العدد الأقصى المسموح به (`maxValue`). كلّ الخانات التّي تملك عددا أكبر من `maxValue` يجب أن تعاد إلى 0. النموذج :

```
void tableMaximum(int table[], int tableSize, int maxValue);
```

التمرين 5

هذا أصعب. أنشئ دالة `sortTable` التي ترتب قيم جدول تصاعديا. كمثال جدول يتكوّن من القيم `{15, 81, 22, 13}`، بعد العملية يصبح : `{13, 15, 22, 81}` !
تفضل النموذج :

```c
void sortTable(int table[], int tableSize);
```

هذا التمرين صعب بقليل عن السابقين لكن القيام به ليس مستحيلا، سيشغلكم بعض الوقت.

> أنشئ ملفا خاصا باسم `tables.c` (مع مرافقه الملف `tables.h` الذي يحتوي النماذج ! بالطبع) يحوي كلّ الدوال التي تقوم بعمليّات على الجداول.

إلى العمل !

ملخّص

- **الجداول** هي عبارة عن مجموعة متغيرات لها نوع واحد مرتبة بجنب بعضها في الذاكرة.
- يجب أن يكون حجم الجدول محدداً قبل ترجمة البرنامج، لا يمكن أن تعتمد على متغيّر.
- الجدول ذو النوع `int` لا يحتوي سوى متغيرات من نوع `int`.
- خانات الجدول مرقّمة عن طريق **الفهارس** ابتداءاً من 0 : `table[0]`، `table[1]`، `table[2]`، إلخ.

الفصل 13

السلاسل المحرفيّة (Strings)

السلاسل المحرفيّة هي اسم صحيح برمجيّاً لتسمية ... النصّ، ببساطة !
السلسلة المحرفيّة هي إذن نصّ يمكننا حفظه على شكل متغيّر في الذاكرة. بهذه الطريقة يمكننا تخزين اسم المستخدم.

كنت قد قلت من قبل أن الحاسوب لا يفهم إلا الأعداد، فما هو السحر الذي يفعله المبرمجون للتعامل مع النصوص ؟ إنّهم ماكرون، سوف ترى !

1.13 النوع char

في هذا الفصل سنعطي أهمية خاصة للنوع `char`. إن كنت تتذكّر جيداً فهذا النوع يسمح بتخزين الأعداد المحصورة بين -127 و 128.

> النوع `char` يسمح بتخزين الأعداد، لكننا غالبا لا نستخدمه في لغة الـC من أجل ذلك. عادة، حتّى لو كان العدد صغيرا، فإنّنا نخزنه في `int`. بالطبع، هذا سيأخذ شيئا أكبر من الذاكرة، لكن في هذه الأيّام، ليست الذاكرة ما ينقص الحواسيب فعلا.

إذا فالنوع `char` مستعمل لتخزين ... "حرف" ! احذر، لقد قلت : حرف واحد.

و لأنّ الذاكرة لا يمكنها تخزين شيء سوى الأعداد، فلقد تمّ اختراع جدول يقوم بالتحويل بين الحروف و الأعداد. هذا الجدول يخبرنا مثلا أنّ العدد 65 مكافئ للحرف A.

لغة C تسمح لنا بالقيام بالتحويل بين الحرف و العدد الموافق له. للحصول على العدد الموافق لحرف، يكفي كتابته بين علامتي تنصيص، هكذا : `'A'`. عند الترجمة، سيتم استبدال `'A'` بالقيمة الموافقة.

فلنجرّب :

```
1  int main(int argc, char *argv[])
2  {
3          char letter = 'A';
4          printf("%d\n", letter );
5          return 0;
6  }
```

نعلم إذن أن الحرف A يمثل بالعدد 65، B بـ66، C بـ67، إلخ.
جرّب بالأحرف الصغيرة و سترى أن القيم مختلفة. في الواقع، الحرف `'a'` ليس مطابقا لـ `'A'`، الحاسوب يقوم بالتفريق بين الحروف الصغيرة و الكبيرة (نقول أنّه يحترم حالة الحرف).

أغلب الحروف "الأساسيّة" مشفّرة بين 0 و 127. يوجد جدول يقوم بالتحويل بين الأعداد و الحروف : الجدول ASCII (ينطق "أسكي"). الموقع AsciiTable.com مشهور لعرض هذا الجدول لكنّه ليس الوحيد، يمكننا أن نجده على ويكيبيديا و مواقع أخرى أيضا.

إظهار محرف

كما نعلم فلإظهار أي شيء على الشاشة نستعمل الدالة `printf`، هذه الدالة قادرة أيضا على إظهار محرف على الشاشة و ذلك باستعمال الرمز : `%c` (c تعني Character) كالتالي :

```
1 int main(int argc, char *argv[])
2 {
3         char letter = 'A';
4         printf("%c\n", letter);
5         return 0;
6 }
```

```
A
```

حسنا، لقد تعلّمنا كيف نظهر حرفا في الشاشة.

يمكننا أيضا أن نطلب من المستعمل أن يقوم بإدخال حرف عن طريق لوحة المفاتيح، و ذلك بالإستعانة بالدالة `scanf`، و هذا بوضع الرمز `%c` دائماً، كالتالي :

```
1 int main(int argc, char *argv[])
2 {
3         char letter = 0;
4         scanf("%c", &letter);
5         printf("%c\n", letter);
6         return 0;
7 }
```

إن كتبت الحرف B فسأتحصّل على :

```
B
B
```

أوّل B هو الّذي كتبته، أمّا الثاني فهو المعروض من طرف `printf`.

هذا تقريبا ما يجب عليك أن تتذكره بخصوص النوع `char`. تذكّر جيدّا :

- النوع `char` يخزّن الأعداد من −128 إلى 127، بينما النوع `unsigned char` يسمح بتخزين الأعداد من 0 إلى 255.
- يستخدم الحاسوب جدولا للتحويل بين الحروف و الأعداد، الجدول ASCII.
- يمكن استخدام `char` لتخزين حرف واحد.
- `'A'` يتمّ استبدالها أثناء الترجمة بالقيمة الموافقة (65 مثلا). نستخدم إذن علامات التنصيص للحصول على قيمة حرف.

2.13 السلاسل المحرفيّة هي جداول من نوع char

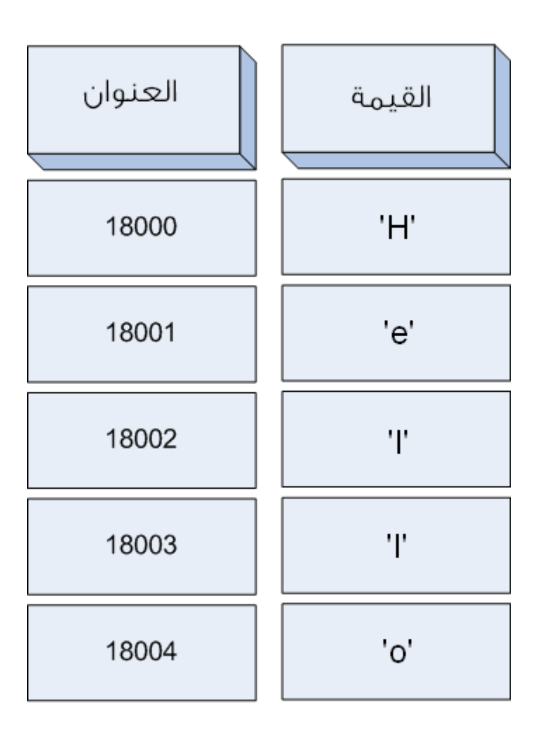
مثلما يشير العنوان. في الواقع، فإن السلسلة المحرفيّة ما هي إلا عبارة عن جدول من نوع `char`. مجرّد جدول بسيط.

إن قمنا بإنشاء جدول :

```
char string[5];
```

و قمنا بوضع الحرف `'H'` في `'string[0]'`، الحرف `'e'` في `'string[1]'` ... فيمكننا تكوين سلسلة محرفيّة، أي نصّ.

المخطط التالي يعطيك فكرة عن كيفيّة تخزين السلسلة في الذاكرة (احذر : في الحقيقة الأمر أصعب بقليل مما هو ظاهر، سأشرح ذلك لاحقا).



كما نرى فهذا جدول يتكون من 5 خانات في الذاكرة ليمثل الكلمة 'Hello'. في المخطط اخترت تمثيل الحروف بين علامتي تنصيص لأبيّن أنه يتمّ تخزين عدد و ليس حرف. في الحقيقة، دائما في الذاكرة، يتمّ تخزين الأعداد الموافقة لهذه الحروف.

سلسلة المحارف لا تحتوي فقط على الحروف، في الواقع المخطط السابق غير كامل ! السلسلة المحرفيّة **تحتوي بالضرورة محرفا خاصّا في النهاية**، يسمّى "محرف نهاية السلسلة". هذا المحرف يكتب `\0`.

> ؟ لماذا يجب أن تنتهي السلسلة المحرفيّة بـ `\0` ؟

ببساطة لكي يعرف الحاسوب أين تنتهي السلسلة. المحرف `\0` يقول : "توقّف، لا يوجد المزيد لقراءته !".

لذلك، كي نخزن الكلمة 'Hello' لا نحتاج إلى جدول من 5 `char` و إنما من 6 !
في كلّ مرة تقوم فيها بإنشاء سلسلة محرفيّة، يجب عليك أن تفكّر في حجز مكان لمحرف نهاية السلسلة. يجب دائما إضافة خانة لتخزين هذا المحرف `\0`، هذا ضروريٌ !

نسيان محرف نهاية السلسلة `\0` هو مصدر أخطاء موجعة في لغة C. لهذا فأنا أكرر هذا التحذير أكثر من مرّة.

المخطط التالي هو الأصحّ في تمثيل السلسلة المحرفيّة 'Hello' في الذاكرة.

العنوان	القيمة
18000	'H'
18001	'e'
18002	'l'
18003	'l'
18004	'o'
18005	'\0'

كما ترى، السلسلة تحوي 6 محارف لا 5، يجب أن يكون الأمر كذلك. السلسلة تنتهي بـ `\0`، محرف نهاية السلسلة يسمح للحاسوب بمعرفة أين تنتهي السلسلة.

اعتبر المحرف `\0` شيئا إيجابيّا لك. بفضله ليس عليك تذكر حجم الجدول الذي خزنته لأنّه يدلّ على مكان توقّف الجدول. يمكنك أن تمرّر جدول `char` دون الحاجة إلى استخدام متغيّر يدلّ على حجمه.
هذا الأمر صالح فقط للسلاسل المحرفيّة، (أي النوع `char*` الذي يمكننا أيضا كتابته على النحو `char[]`). بالنسبة للأنواع الأخرى من الجداول، عليك أن تحفظ حجم الجدول في مكان ما.

إنشاء و تهيئة سلسلة محرفيّة

إن أردنا إنشاء جدول `string` يحتوي النصّ 'Hello'، يمكننا استعمال الطريقة اليدويّة و لكنّها غير فعّالة :

```
char string[6]; // A table of  6 chars used to store H—e—l—l—o + \0
string[0] = 'H';
string[1] = 'e';
string[2] = 'l';
string[3] = 'l';
string[4] = 'o';
string[5] = '\0';
```

هذه الطريقة تعمل. يمكننا التحقق من ذلك باستعمال `printf`.

لاستخدام `printf` يجب أن نستعمل الرمز `%s` (s تعني String، أي "سلسلة محارف" بالإنجليزية). هذه هي الشفرة الكاملة التي تنشئ السلسلة 'Hello' في الذاكرة :

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        char string[6]; // A table of 6 chars used to store H—e—l—l—o + \0
        // Initializing the string (writing the letters one by one in the
            memory)
        string[0] = 'H';
        string[1] = 'e';
        string[2] = 'l';
        string[3] = 'l';
        string[4] = 'o';
        string[5] = '\0';
        // Displaying the string thanks to the %s in the function printf
        printf("%s", string);
        return 0;
}
```

النتيجة :

```
Hello
```

تلاحظ أن القيام بتخزين النص حرفا بحرف في الجدول `string` أمر متعب للغاية. لتهيئة سلسلة محرفيّة توجد، لحسن الحظ، طريقة أبسط بكثير :

```
int main(int argc, char *argv[])
{
        char string[] = "Hello"; // The size of the table is automatically
            calculated
        printf("%s", string);
        return 0;
}
```

```
Hello
```

كما تلاحظ في السطر الأول ترى أنني أنشأت متغيّرا من نوع `char[]`، كان بإمكاني كتابة `char*` أيضاً، النتيجة ستكون نفسها.

عندما تكتب بين علامتي اقتباس (" ") النص الذين تريد تخزينه في الجدول، يقوم الحاسوب بحساب الحجم اللازم. أي أنّه سيحسب عدد الحروف و يضيف 1 من أجل المحرف `\0`. يبدأ بعدها في تخزين حروف الكلمة 'Hello' واحدا واحدا في الذاكرة و في النهاية يضيف `\0` تماما كما فعلنا يدويّا منذ قليل.
باختصار، هذا أمر عمليّ أكثر.

و مع ذلك فهناك مشكل : هذا الأمر لا يعمل إلّا مع التهيئة ! لاحقا في الشفرة لا يمكنك كتابة :

```
string = "Hello";
```

هذه التقنيّة محجوزة للتهيئة فقط. بعد هذا، يجب أن نقوم بالتعديل على المحارف يدويّا في الذاكرة واحداً واحداً كما فعلنا في البداية.

إدخال سلسلة محرفيّة عن طريق scanf

يمكننا حفظ سلسلة محرفيّة مُدخَلة من طرف المستخدم عن طريق `scanf`، باستخدام الرمز `%s`.
مشكل وحيد : لا يمكنك معرفة كم محرفا سيقوم المستخدم بإدخاله. إن طلبت منه اسمه الأوّل، فيمكن أن يسمّى Luc (3 محارف)، و لكن من الذي سيضمن أن اسمه لن يكون Jean-Edouard (عدد أكبر بكثير من المحارف) ؟

من أجل هذا، لا يوجد 36 حلّا. يجب إنشاء جدول من `char` كبير جدّا، كبير بما يكفي لتخزين الاسم. سنقوم إذن بإنشاء `char[100]`. قد تشعر بأنّ هذا إهدار للذاكرة، لكن تذكّر مرّة أخرى بأنّ المكان في الذاكرة ليس الشيء الذي ينقصنا (كما أنّه توجد برامج تهدر الذاكرة بطريقة أسوء بكثير من هذه !).

```
int main(int argc, char *argv[])
{
        char firstName[100];
        printf("What's your name ? ");
        scanf("%s", firstName);
        printf("Hello %s, nice to meet you !", firstName);
        return 0;
}
```

```
What's your name ? Mateo21
Hello Mateo21, nice to meet you !
```

3.13 دوال التعامل مع السلاسل المحرفيّة

السلاسل المحرفيّة مستعملة بكثرة. فكلّ الكلمات، كلّ النصوص التي تراها على الشاشة هي في الواقع جداول `char` في الذاكرة و تعمل كما شرحت لك من قبل.
للمساعدة في التعامل مع السلاسل المحرفيّة، توجد المكتبة `string.h` التي تحتوي على كم كبير من الدوال التي تقوم بالحسابات على السلاسل المحرفيّة.

لا يمكنني أن أشرحها لك كلّها هنا، سيتطلّب الأمر وقتا طويلا كما أنّها ليست كلّها ضروريّة.
سأعلّمك الأساسيّة منها و الّتي ستحتاجها حتما لاحقا.

فكّر في تضمين `string.h`

حتّى لو بدا لك هذا الأمر بديهيّا، فأنا أفضّل أن أتحدّث عنه مرّة أخرى : بما أنّنا سنستخدم مكتبة جديدة تسمّى `string.h`، فيجب أن تضمّنها في أعلى ملفّات `.c` حيثما تحتاجها :

```
1 #include <string.h>
```

إن لم تقم بهذا فإن المترجم لن يتعرف على الدوال التي سأعرضها لك لأنّه لا يملك نماذجها، و بالتالي فستتوقف الترجمة.
باختصار، لا تنس تضمين هذه المكتبة في كلّ مرّة تستخدم فيها دوال التعامل مع النصوص.

`strlen` : حساب طول سلسلة محرفيّة

`strlen` تقوم بحساب طول السلسلة المحرفيّة (دون حساب المحرف `\0`).
يجب أن تعطيها معاملا وحيدا : السلسلة المحرفيّة. هذه الدالة تُرجع طول السلسلة.

الآن بما أنّك تعرف مالّذي يعنيه النموذج، سأعطيك نموذج الدوال الّتي أكلّمك عنها. المبرمجون يعتبرونه كـ"دليل استخدام" للدالّة. هذا نموذج الدالة :

```
1 size_t strlen(const char* string);
```

> `size_t` هو نوع خاص يدلّ على أنّ الدالة تعيد عددا يمثّل طولا. هو ليس نوعاً قاعديا مثل `int`، `float`، `char` و إنما هو نوع "مُختَرَع". سنتعلم نحن كيف نقوم بإنشاء أنواعنا الخاصة في فصول لاحقة. حاليّا، سنكتفي بتخزين النتيجة التي تعيدها `strlen` في متغيّر من نوع `int` (سيقوم الحاسوب بالتحويل تلقائيّا من `size_t` إلى `int`). يفترض أننا نقوم بتخزين هذه القيمة في متغيّر من نوع `size_t`، لكن عمليّا `int` كافٍ لهذا.

الدالّة تأخذ معاملا من نوع `const char*`. الـ `const` (التي تعني ثابت، تذكّر) تعني أن الدالة "ستَمتَنِع" عن تغير السلسلة. عندما ترى `const`، فستعلم أن المتغيَر لا يتمّ تعديله من طرف الدالّة، بل قراءته فقط.

فلنجرّب الدالّة `strlen`

```c
int main(int argc, char *argv[])
{
        char string[] = "Hello";
        int stringLength = 0;
        // We put the length of string in stringLength
         stringLength = strlen(string);
        // We display the length of string
        printf("The string %s contains %d characters", string, stringLength);
        return 0;
}
```

```
The string Hello contains 5 characters
```

هذه الدالة سهلة الكتابة، يكفي القيام بحلقة على جدول `char` حتى تصل إلى المحرف `\0`. يوجد عدّاد تتمّ زيادته في كل دورة من الحلقة، و هذا العدّاد يتمّ إعادته في النهاية.

هذا جعلني أريد كتابة شفرة شبيهة بتلك الخاصّة بـ `strlen`. هذا سيمكّنك من أن تفهم جيّدا كيف تعمل هذه الدالّة :

```c
int stringLen(const char* string);
int main(int argc, char *argv[])
{
        char string[] = "Hello";
        int stringLength = 0;
        // We put the length of string in stringLength
        stringLength = stringLen(string);
        // We display the length of string
        printf("The string %s contains %d characters", string, stringLength);
        return 0;
}

int stringLen(const char* string)
{
        int charactersCount = 0;
        char currentCharacter = 0;
        do
        {
                currentCharacter = string[charactersCount];
                charactersCount++;
        }
        while(currentCharacter != '\0'); // We loop while we didn't reach \0
        charactersCount--; // We decrement by 1 in order to not count \0
        return charactersCount;
}
```

هذه الدالة `stringLen` تقوم بحلقة على الجدول `string`. تقوم في كل مرة بتخزين المحرف الحالي في محرف مُساعد سميناه `currentCharacter`، ما إن يكون المحرف الحالي هو `\0`، نخرج من الحلقة.

في النهاية نقوم بإنقاص 1 من المجموع، لكي لا نحسب `\0`. قبل نهاية الدالة، نقوم بإرجاع الطول الذي حسبناه أي `charactersCount`.

`strcpy` : نسخ سلسلة محرفيّة في أخرى

الدالة `strcpy` (و التي تعني "String copy") تسمح بنسخ سلسلة محرفيّة إلى داخل أخرى.
نموذج الدالّة :

```
1 char* strcpy(char* stringCopy, const char* stringToCopy);
```

الدالّة تأخذ معاملين :

- `stringCopy` : مؤشّر نحو `char*` (جدول `char`). في هذا الجدول يتمّ لصق السلسلة.
- `stringToCopy` : مؤشّر نحو جدول آخر من `char`. هذه السلسلة يتمّ نسخها إلى `stringCopy`.

الدالّة تعيد مؤشّرا نحو `stringCopy`، و هو غير مفيد جدّا. عادة، لا نحفظ ما تعيده هذه الدالّة. فلنجرّبها :

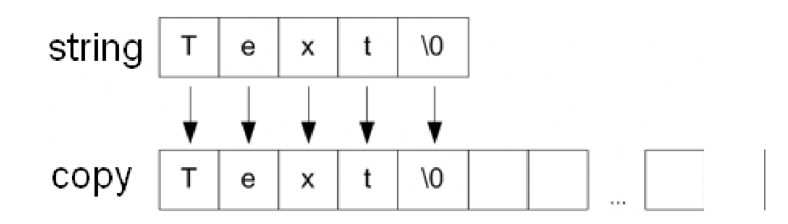
```
1 int main(int argc, char *argv[])
2 {
3         char string[] = "Text", copy[100] = {0};
4         strcpy(copy, string); // We copy "string" in "copy"
5         // If everything is okay, the copy must contain the same thing as
             string
6         printf("string is : %s\n", string);
7         printf("copy  is : %s\n", copy);
8         return 0;
9 }
```

```
string is : Text
copy is : Text
```

نلاحظ أنّ قيمة `string` هي "Text". حتّى الآن، الأمر عاديّ.
بالمقابل، نرى أيضا أنّ المتغيّر `copy`، الذي كان فارغا في البداية، قد تمّ ملؤه بمحتوى `string`. لقد تمّ فعلا نسخ السلسلة في `copy`.

> ! تأكّد أنّ السلسلة `copy` كبيرة كفاية لاحتواء محتوى `string`. إن قمت في هذا المثال بتعريف `copy[5]` (و الذي هو غير كاف لأنه لن يبق مكان لـ `\0`)، الدالّة `strcpy` "ستخرج عن الذاكرة" و ربّما ستوقف البرنامج عن العمل. تجنّب ذلك، إلّا إذا كنت تريد تعطيل برنامجك.

النسخ يمكن تمثيله كالتالي :



كلّ محرف من `string` يتمّ وضعه في `copy`.
السلسلة المحرفيّة `copy` تحوي عددا كبيرا من المحارف غير المستعملة، قد لاحظت هذا. أعطيته الحجم 100 من باب الحماية، لكنّ الحجم 6 كان كافيا. الفائدة من إنشاء جدول أكبر هي أنّه بهذه الطريقة، السلسلة المحرفيّة `string` تكون قادرة على احتواء سلاسل أخرى قد تكون أكبر في بقيّة البرنامج.

`strcat` : وصل سلسلتين محرفيّتين

هذه الدالة تضيف سلسلة محرفيّة إلى نهاية الأخرى. نسمّي هذه العملية بالوصل (Concatenation).
فلنفرض أن لدينا المتغيّرات التالية :

- `string1 = "Hello "`
- `string2 = "Mateo21"`

إن وصلت `string1` في `string2` فـ `string2` ستصبح `"Hello Mateo21"`. أمّا بالنسبة لـ `string2`، فلن تتغيّر و ستكون دائما `"Mateo21"`. فقط `string1` التي تتغيّر.

هذا تماما ما تفعله `strcat`، هذا هو نموذجها :

```
1 char* strcat(char* string1, const char* string2);
```

كما يمكنك أن ترى، `string2` غير قابل للتعديل لأنّه تمّ التصريح عنه كثابت في نموذج الدالّة.
الدالّة تعيد مؤشّرا نحو `string1`، و مثلما هو الحال مع `strcpy`، فهذا لا يصلح لشيء كبير في حالتنا هذه : يمكننا إذن تجاهل ما تعيده الدالّة.

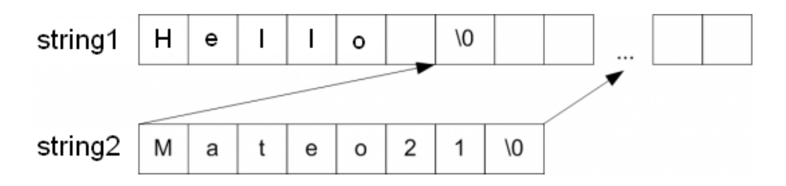
الدالّة تضيف إلى `string1` محتوى `string2`. فلنرى هذا عن قرب :

```
1  int main(int argc, char *argv[])
2  {
3          char string1[100] = "Hello ", string2[] = "Mateo21";
4          strcat(string1, string2); // We concatenate string2 to string1
5          // If everything is okay, string1 is equal to "Hello Mateo21"
6           printf("string1 is : %s\n", string1);
7          // string2 didn't change :
8          printf("string2 is as always : %s\n", string2);
9          return 0;
10 }
```

```
string1 is : Hello Mateo21
string2 is as always : Mateo21
```

تأكّد دائما أنّ `string1` كبيرة بما فيه الكفاية لكي نستطيع إضافة محتوى `string2` إليها، و إلّا فستحدث خروجا عن الذاكرة و قد يتسبب في تعطّل البرنامج.
لهذا السبب قمت بتعريف `string2` بحجم 100. بينما تركت الحاسوب يحسب حجم `string1` تلقائيّاً (هذا يعني أنني لم أحدد الحجم) لأنّ هذه السلسلة لا يتمّ تعديلها، فلا حاجة إذن لجعلها أكبر من اللازم.

المخطط التالي يلخّص عمليّة الوصل.



الجدول `string2` تمّت إضافته إلى نهاية `string1` (الذي يحوي 100 خانة). الـ`\0` الخاص بـ`string1` تمّ حذفه (في الواقع تمّ استبداله بـM من Mateo21). في الواقع، لا يجب ترك `\0` في وسط سلسلة محرفيّة، و إلّا فسيتمّ "قطعها" في المنتصف ! لا نضع `\0` إلّا في نهاية سلسلة محرفيّة، فقط عندما ننهيها.

strcmp : مقارنة سلسلتين محرفيّتين

`strcmp` تقارن سلسلتين فيما بينهما. هذا هو نموذجها :

```
1 int strcmp(const char* string1, const char* string2);
```

المتغيّران `string1` و `string2` يتمّ مقارنتهما. كما تلاحظ، لا يتمّ تعديل أيّ منهما لأنّه تمّ التصريح عنهما كثوابت.

إنه من الضروري أن نقوم باسترجاع ما تعيده إلينا الدالة. في الواقع، `strcmp` تعيد :

- 0 إن كانت السلسلتان متطابقتين.
- قيمة أخرى (موجبة أو سالبة) إن كانت السلسلتان مختلفتين.

أعلم أنّه كان من المنطقيّ أكثر أن تعيد الدالّة 1 إن كانت السلسلتين متطابقتين لكي نقول "صحيح" (تذكّر المتغيّرات المنطقيّة). السبب بسيط : الدالّة تقارن قيم المحارف من كلّ سلسلة واحدا واحدا. إن كانت كلّ المحارف متطابقة فستعيد 0. إن كانت محارف `string1` أكبر من محارف `string2`، فالدالّة تعيد عددا موجبا. في الحالة المعاكسة تعيد عددا سالبا. عمليّاً، نستخدم `strcmp` كثيرا للتحقّق من أنّ سلسلتين متطابقتان أم لا.

هذه شفرة التجريب الخاصّة بي :

```c
int main(int argc, char *argv[])
{
        char string1[] = "Test text", string2[] = "Test text";
        if (strcmp(string1, string2) == 0) // If the strings are equal
        {
                printf("The strings are identical\n");
        }
         else
        {
                printf("The strings are different\n");
         }
        return 0;
}
```

```
The strings are identical
```

بما أنّ السلسلتين متطابقتين، فالدالّة `strcmp` أعادت 0.
لاحظ أنّه كان بإمكاني تخزين ما أعادته الدالّة في متغيّر من نوع `int` . لكنّ ذلك ليس ضروريّا، فيمكننا وضعها مباشرة داخل `if` كما فعلت.

ليس لديّ ما أضيفه فيما يتعلّق بهذه الدالّة. هي بسيطة الاستخدام، لكنّ الشيء الوحيد الذي لا يجب نسيانه هو أنّ 0 يعني "متطابق" و أي قيمة أخرى تعني "مختلف". هذا هو مصدر الأخطاء الوحيد هنا.

strchr : البحث عن محرف

الدالّة `strchr` تبحث عن محرف في سلسلة محرفيّة. نموذجها :

```c
char* strchr(const char* string, int characterToFind);
```

الدالّة تأخذ معاملين :

- `string` : السلسلة التي يتمّ البحث فيها.
- `characterToFind` : الحرف الّذي نبحث عنه في السلسلة.

تلاحظ أنّ `characterToFind` من نوع `int` و ليس من نوع `char` . هذا ليس مشكلا فعلا لأنّه في الواقع الحرف هو عدد. على الرغم من ذلك، نفضّل استخدام `char` على `int` لتخزين المحارف في الذاكرة.

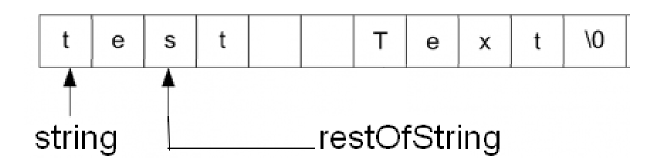
الدالة تقوم بإرجاع مؤشر نحو أوّل ظهور للمحرف الذي وجدته في السلسلة، أي أنها ترجع عنوانه في الذاكرة. إن لم تجد شيئا فستعيد `NULL` .
في المثال التالي، سأسترجع هذا المؤشّر في `restOfString` .

```
1  int main(int argc, char *argv[])
2  {
3          char string[] = "Test text", *restOfString = NULL;
4          restOfString = strchr(string, 's');
5          if (restOfString != NULL) // If we found something
6          {
7                  printf("This is the rest of the string after the first s:%s",
                       restOfString);
8           }
9    return 0;
10 }
```

```
This is the rest of the string after the first s: st text
```

هل فهمت ما حصل ؟ إن الأمر خاصّ قليلا.
في الحقيقة، إن `restOfString` هو مؤشر مثل `string`، إلّا أنّ `string` يؤشّر على الحرف الأوّل (أي `'T'`) بينما `restOfString` يؤشّر على أوّل محرف `'s'` موجود في `string`.

المخطّط التالي يوضّح أين يؤشّر كلّ مؤشر :



عندما أقوم بـ `printf` لـ `restOfString`، فمن العاديّ أن يتمّ عرض "st Text". الدالّة `printf` تعرض كلّ المحارف التي تلقاها (s،t،T،e،x،t) حتّى الوصول إلى `\0` الذي يعلمها بنهاية السلسلة.

نسخة مختلفة

توجد دالّة `strrchr` مطابقة تماما لـ `strchr` باستثناء أنها تعيد مؤشّرا على آخر ظهور **للمحرف** الموجود في السلسلة بدلا من الأوّل.

strpbrk : أوّل محرف في القائمة

هذه الدالّة تشبه كثيرا السابقة. هذه تبحث عن محرف من بين تلك التي تعطيها على شكل سلسلة محرفيّة، بخلاف `strchr` التي لا يمكنها البحث عن سوى محرف وحيد في المرّة.

مثلا، إن أعطيناها السلسلة `"xes"` و بحثنا في `"Test text"`، فالدالّة تعيد مؤشرا نحو أول تكرار لأحد المحارف التي وجدتها. في هذه الحالة، أوّل محرف من `"xes"` التّي ستجده في `"Test text"`، هو الـ `'e'`، إذن `strpbrk` تعيد مؤشرا على `'e'`.

النموذج :

```
char* strpbrk(const char* string, const char* charactersToFind);
```

فلنجرّب الدالّة :

```
int main(int argc, char *argv[])
{
        char *restOfString;
        // We search for the 1st occurrence of x, e or s in "Test text"
        restOfString= strpbrk("Test text", "xes");
        if (restOfString != NULL)
         {
                printf("This is the rest of the string starting by the first
                    occurrence of the characters found : %s", restOfString);
        }
        return 0;
}
```

```
This is the rest of the string starting by the first occurrence of the
    characters found : est text
```

في هذا المثال، القيم المراد ارسالها إلى الدالّة مباشرة (بين علامتي اقتباس). لسنا مجبرين على استخدام متغيّر في كلّ المرّات، يمكننا كتابة السلسلة مباشرة.
يجب تذكّر هذه القاعدة البسيطة :

- إذا استخدمت علامتي الاقتباس `" "`، فهذا يعني **سلسلة محرفيّة**.
- إذا استخدمت علامتي التنصيص `' '`، فهذا يعني **محرفا**.

strstr : البحث عن سلسلة محرفيّة في أخرى

هذه الدالّة تبحث عن أوّل ظهور لسلسلة محرفيّة داخل أخرى.
نموذجها :

```
char* strstr(const char* string, const char* stringToFind);
```

النموذج مشابه لذلك الخاص بـ`strpbrk`، لكن احذر من الخلط : `strpbrk` تبحث عن واحد من المحارف، بينما `strstr` تبحث عن كلّ السلسلة.

مثال :

```
int main(int argc, char *argv[])
{
        char *restOfString;
        // We search for the 1st occurrence of "text" in "Test text" :
        restOfString = strstr("Test text", "text");
```

```
6         if (restOfString != NULL)
7          {
8                 printf("First occurrence of text in Test text : %s\n",
                     restOfString);
9         }
10        return 0;
11 }
```

```
First occurrence of text in Test text : text
```

الدالّة `strstr` تبحث عن السلسلة "text" في "Test text".
كغيرها، تعيد مؤشّرا عندما تجد ما تبحث عنه. و تعيد `NULL` إن لم تجد شيئا.

حتّى الآن، أنا أقوم بعرض السلسلة عن طريق المؤشّر الذي تعيده الدالّة. عمليّا، هذا غير مهمّ جدّا. يمكنك فقط القيام بـ`if (result != NULL)` لمعرفة إن مان البحث قد أعاد أيّ شيء، و تُظهر "النصّ الذي تبحث عنه قد تمّ العثور عليه".

sprintf : الكتابة في سلسلة محرفيّة

> هذه الدالّة موجودة في `stdio.h` بخلاف الدوال التي درسناها إلى حدّ الآن، و التي كانت في `string.h`

هذا الاسم يذكّرك بشيء ما. هذه الدالّة مشابهة إلى حدّ كبير لـ`printf` التي تعرفها، و لكن بدل الكتابة على الشاشة، `sprintf` تكتب في ... سلسلة محرفيّة ! و من هذا اسمها الذي يبدأ بـ"s" من "string" (سلسلة محرفيّة بالإنجليزيّة).

إنّها دالّة عمليّة جدّا لتنسيق سلسلة محرفيّة. مثال صغير :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char *argv[])
4  {
5          char string[100];
6          int age = 15;
7          // We write "You have 15 years" in string
8          sprintf(string, "You have %d years !", age);
9          // We display string to check its content
10         printf("%s", string);
11         return 0;
12 }
```

```
You have 15 years !
```

تُستخدم بنفس طريقة `printf` باستثناء أنّه يجب إعطائها كمعامل أوّل مؤشّرا نحو السلسلة التي يجب أن تستقبل النصّ.

في هذا المثال، أكتب في `string` "You have %d years"، حيث يتمّ استبدال `%d` بمحتوى المتغيّر `age`. كلّ قواعد `printf` تطبّق، يمكنك إذا أردت أم تضع `%s` لإدراج سلاسل أخرى داخل سلسلتك !

كالعادة، تأكّد أن سلسلتك كبيرة كفاية لاحتواء النصّ الذي سترسله لها `sprintf`. و إلّا، فقد يحدث تجاوز في الذاكرة و بالتالي تعطّل برنامجك.

ملخّص

- الحاسوب لا يجيد التعامل مع النصوص، هو لا يعرف إلّا الأعداد. لإصلاح هذا المشكل، تمّ ربط كلّ حرف بعدد موافق له في جدول يسمّى **جدول ASCII**.
- النوع `char` يستخدم لتخزين حرف واحد. يخزّن في الحقيقة عددا، لكنّ هذا العدد تمّ ترجمته إلى حرف من طرف الحاسوب أثناء العرض.
- لإنشاء كلمة أو جملة، علينا بناء **سلسلة محرفيّة**. من أجل هذا، نستخدم **جدول char**.
- كلّ سلسلة محرفيّة تنتهي بحرف خاصّ يكتب `\0` يعني "نهاية السلسلة".
- توجد الكثير من الدوال الجاهزة للتعامل مع السلاسل المحرفيّة في **المكتبة string**. يجب تضمين `string.h` لاستخدامها.

الفصل 14

المعالج القبلي (Preprocessor)

بعد كل المعلومات المتعبة التي تلقّيتها في الفصول حول الجداول، النصوص و المؤشرات، فسنقوم بالتوقف قليلا. لقد تعلّمت أشياء جديدة كثيرة في الفصول السابقة، لن يكون لديّ مانع من نسترجع أنفاسنا قليلا.

هذا الفصل يتحدّث عن المعالج القبلي، هذا البرنامج الّذي يعمل مباشرة قبل الترجمة.
لا تخطئ : المعلومات التي به ستكون مهمّة لك. لكنّها ستكون أقل تعقيداً من الّتي تعلّمتها مؤخّراً.

1.14 الـinclude

كما شرحت لك في الفصول الأولى من الكتاب، نجد في الشفرات المصدريّة سطورا خاصّة تسمّى **بتوجيهات المعالج القبلي** (Preprocessor directives).
هذه السطور لديها الخاصيّة التالية : تبدأ دائما بالرمز `#`. لذا فمن السهل التعرّف عليها.

التوجيهة الوحيدة التي رأيناها لحدّ الآن هي `#include`.
هذه التوجيهة تسمح لنا بتضمين محتوى ملف في آخر. قلت لك هذا من قبل.
نحن نحتاجها في تضمين الملفات ذات الصيغة `.h` كملفات `.h` الخاصّة بالمكتبات (`stdlib.h`، `stdio.h`، `string.h`، `math.h`...)، و أيضاً ملفات `.h` الخاصّة بنا.

لنضمّن ملفاً ذو صيغة `.h` موجوداً في نفس المجلّد الذي ثبتنا فيه الـIDE (أي البيئة التطويرية كالـCode::Blocks مثلا)، نستعمل علامات الترتيب `< >` كالتالي :

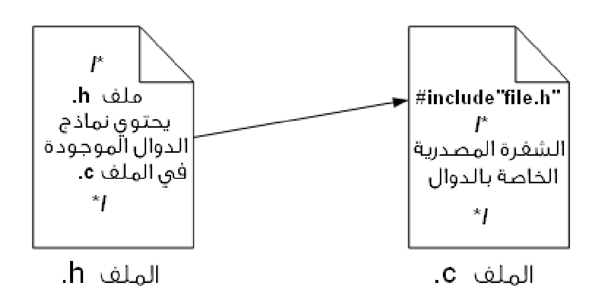
```c
#include <stdlib.h>
```

بينما لتضمين ملفّ `.h` موجود في المجلّد الذي به مشروعنا، فسنقوم بذلك باستخدام علامتي الترتيب كالتالي :

```c
#include "myfile.h"
```

في الحقيقة، المعالج القبلي يتمّ تشغيله قبل الترجمة. يبحث في كلّ ملفاتك عن توجيهات المعالج القبلي، تلك الأسطر المشهورة التي تبدأ بـ`#`.
عندما يجد التوجيهة `#include`، يقوم بإدراج محتوى الملفّ في مكان وجود `#include`.

افترض أن لديّ ملفّا `file.c` يحتوي الشفرة الخاصة بالدوال التي كتبتها، و لدي ملف `file.h` يحتوي نماذج الدوال التي هي موجودة بالملف `file.c`، يمكن تلخيص ذلك بالمخطط التالي.



كل محتوى الملف `file.h` سيتم وضعه داخل الملف `file.c` في مكان التوجيهة `#include "file.h"`.

تخيّل أن لدينا في الملف `file.c` التالي :

```c
#include "file.h"

int myFunction(int something, double stupid)
{
  /* The code of the function */
}
void anotherFunction(int value)
{
  /* The code of the function */
}
```

و في الملف `file.h` :

```c
int myFunction(int something, double stupid);
void anotherFunction(int value);
```

عندما يمر المعالج القبلي بهذه الشفرة، قبل أن تتم ترجمة الملف `file.c`، سيضع كما قلت محتوى الملف `file.h` في الملف `file.c`. في النهاية، يعني أن الملف `file.c` قُبيل الترجمة سيحتوي التالي :

```c
int myFunction(int something, double stupid);
void anotherFunction(int value);

int myFunction(int something, double stupid)
{
  /* The code of the function */
}
void anotherFunction(int value)
{
  /* The code of the function */
}
```

محتوى `.h` تمّ إدخاله مكان `#include`.

هذا ليس بالأمر المعقد لفهمه، و لعلّ بعض القراء يشكك في أن الأمر يحصل بهذه الطريقة.
مع هذه الشروحات الإضافيّة، أتمنّى أنّ يوافقني الجميع. الـ`#include` لا تفعل أي شيء سوى إحضار محتوى ملف و تضمينه في آخر، من المهمّ فهم هذا الأمر جيّداً.

> إن كنّا قد قررنا وضع النماذج في ملفّات `.h` بدل ملفّات `.c`، فهذا من المبدأ. بالطبع، كان بإمكاننا وضع نماذج الدوال في أعلى الملفات `.c` بأنفسنا (قد نفعل هذا أحيانا في بعض البرامج الصغيرة)، لكن لأسباب تنظيميّة، من المنصوح به جدّا وضع النماذج في ملفّات `.h`. عندما يكبر برنامجك و يصبح لديك الكثير من ملفّات `.c` يعتمدون على نفس `.h`، ستكون سعيدا لأنّك لن تظطرّ إلى نسخ و لصق النماذج الخاصّة بنفس الدوال عدّة مرّات!

2.14 الـdefine

سنتعرف الآن على توجيهة معالج جديدة و هي `#define`.

هذه التوجيهة تسمح بالتصريح عن **ثابت معالج قبلي**. هذا يسمح بإرفاق قيمة بعبارة.
إليك مثالا :

```
1 #define INITIAL_NUMBER_OF_LIVES 3
```

يجب أن تكتب بالترتيب :

- الـ `define`.
- الكلمة التي تريد ربط القيمة بها.
- قيمة الكلمة.

احذر : رغم التشابه (خصوصا في الاسم الذي اعتدنا كتابته بحروف كبيرة)، فهذه مختلفة كثيرا عن الثوابت التي تعلّمناها حتّى الآن، مثل :

```
1 const int INITIAL_NUMBER_OF_LIVES = 3;
```

الثوابت تأخذ حيّزا في الذاكرة. حتى و إن لم تتغير قيمتها فإن العدد 3 مخزّن في مكان ما من الذاكرة. هذا ليس هو الحال مع ثوابت المعالج القبلي!

كيف تعمل ؟ في الواقع، الـ`#define` تستبدل في شفرتك المصدريّة كلّ الكلمات بقيمتهم الموافقة. هذا تقريبا مثل عمليّة البحث و الاستبدال (Search / Replace) الموجودة في برنامج Word مثلا. إذن السطر :

```
1 #define INITIAL_NUMBER_OF_LIVES 3
```

يستبدل في الملف كلّ `INITIAL_NUMBER_OF_LIVES` بالرقم 3.

هذا مثال على ملف `.c` قبل مرور المعالج القبلي :

```
#define INITIAL_NUMBER_OF_LIVES 3
int main(int argc, char *argv[])
{
        int lives = INITIAL_NUMBER_OF_LIVES;
  /* Code ... */
```

و بعدما يمرّ المعالج القبلي :

```
int main(int argc, char *argv[])
{
        int lives = 3;
  /* Code ... */
```

قبل الترجمة، كلّ `#define` يتمّ استبدالها بالقيمة الموافقة. المترجم "يرى" الملفّ بعد مرور المعالج القبلي، حيث تكون الاستبدالات قد تمت.

ما الفائدة بالنسبة للثوابت التي رأيناها حتّى الآن ؟

كما قلت لك، هي لا تأخذ مكانا في الذاكرة. هذا منطقيّ، نظرا لأنّه عند الترجمة لا يتبقّى سوى الأرقام في الشفرة المصدريّة.

توجد فائدة أخرى و هي أنّ الاستبدال يتمّ في كامل الملف حيث توجد `#define`. إن قمت بتعريف ثابت في الذاكرة داخل دالّة، فلن يكون صالحا إلّا داخل تلك الدالّة، ثم يتمّ حذفه بعد نهايتها. بينما بالنسبة لـ `#define` فإنها تُطبّق على كلّ دوال الملف، و هذا قد يكون عمليّا جدّا في بعض الحالات.

هل من مثال واقعيّ لاستخدام `#define` ؟
هذا ما لن نتأخّر عن فعله. عندما تفتح نافذة في C، قد تحتاج إلى تعريف ثوابت المعالج القبلي لتحديد أبعاد النافذة :

```
#define WINDOW_WIDTH 800
#define WINDOW_HEIGTH 600
```

الفائدة هي أنّه إن أردت تغيير حجم الواجهة (لأنّها تبدو لك صغيرة جدّا)، فيكفي أن تغير `#define` و تعيد ترجمة الشفرة.

لاحظ أنّ `#define` تكون عادة في ملفات `.h` مع نماذج الدوال (بإمكانك أن ترى `.h` الخاصّة بالمكتبات مثل `stdlib.h`، ستجد الكثير من `#define`).
`#define` إذن هي "مسهّلات وصول"، يمكنك تعديل حجم نافذة عن طريق تعديل `#define` بدل الذهاب للبحث في الدوال عن الموضع الذي تفتح فيه النافذة لتعديل الأبعاد. هذا ربح وقت للمبرمج.

كملخّص، ثوابت المعالج القبلي تسمح بـ"إعداد" برنامجك قبل ترجمته. إنّها أشبه بطريقة إعدادات صغيرة.

الـdefine من أجل حجم جدول

نستخدم كثيرا `#define` من أجل تعريف حجم الجداول. نكتب مثلا :

```
#define MAX_SIZE 1000
int main(int argc, char *argv[])
{
        char string1[MAX_SIZE], string2[MAX_SIZE];
        // ...
```

؟ و لكن ... كنت أعتقد أنّه لا يمكننا وضع متغيّر أو ثابت بين القوسين المربعين أثناء تعريف جدول ؟

نعم هذا صحيح، لكنّ `MAX_SIZE` ليس متغيّرا و لا ثابتا. في الواقع لقد قلت لك، المعالج القبلي يحوّل الملف قبل الترجمة إلى :

```
int main(int argc, char *argv[])
{
        char string1[1000], string2[1000];
        // ...
```

و هذا شيء صحيح.

بتعريف `MAX_SIZE` بهذه الطريقة، يمكنك استخدامها لإنشاء جداول ذات حجوم محدّدة. إذا صارت في المستقبل غير كافية، فليس عليك سوى تعديل سطر `#define`، إعادة الترجمة، و جداول `char` تأخذ القيمة الجديدة التّي حددتها.

الحسابات في الـdefine

من الممكن القيام بحسابات صغيرة في الـ`#define`.
مثلا، هذه الشفرة تنشئ ثابتا `WINDOW_WIDTH`، و آخر `WINDOW_HEIGHT`، ثمّ ثالثا `PIXELS_NUMBER`، الذي يحوي عدد البيكسلز المعروضة داخل النافذة (الحساب بسيط : العرض × الطول).

```
#define WINDOW_WIDTH 800
#define WINDOW_HEIGHT 600
#define PIXELS_NUMBER (WINDOW_WIDTH * WINDOW_HEIGHT)
```

قيمة `PIXELS_NUMBER` يتمّ استبدالها قبل الترجمة بالشفرة التالية :
`(WINDOW_WIDTH * WINDOW_HEIGHT)`، أي (800*600)، و تعطينا 480000. ضع دائما حساباتك بين قوسين كما أفعل من باب الاحتياط لكي تعزل العمليّة.

يمكنك القيام بكل العمليات القاعدية التي تعرفها : جمع (+)، طرح (-)، ضرب (*)، قسمة (/)، ترديد (%).

الثوابت مسبقة التعريف

بالإضافة إلى الثوابت التي أنت عرّفتها، فإنه توجد ثوابت معرّفة من قِبَل المعالج القبلي.

كل من هذه الثوابت تبدأ و تنتهي برمزي underscore `_` (تجده في لوحة المفاتيح تحت الرقم 8 أعلى اللوحة بالنسبة للتخطيط AZERTY).

- `__LINE__` : يعطي رقم السطر الحالي من الشفرة.
- `__FILE__` : يعطي اسم الملف الحالي.
- `__DATE__` : يعطي تاريخ ترجمة الشفرة.
- `__TIME__` : تعطي وقت ترجمة الشفرة.

قد تكون هذه الثوابت مفيدة لمعالجة الأخطاء، مثال :

```c
printf("Error in the line n° %d of the file %s\n", __LINE__, __FILE__);
printf("This file has been compiled on %s at %s\n", __DATE__, __TIME__);
```

```
Error in the line n° 9 of the file main.c
This file has been compiled on 13 Jan 2006 at 19:21:10
```

المعرّفات البسيطة

إنه من الممكن أن نكتب بكل بساطة :

```c
#define CONSTANT
```

دون إعطاء القيمة.
هذا يعني للمعالج القبلي أنّ الكلمة `CONSTANT` معرّفة، بكلّ بساطة. ليست لها قيمة لكنّها "موجودة".

> ؟ ما الفائدة من ذلك ؟

الفائدة قد لا تبدو واضحة كما كان الأمر في السابق، لكن لهذا فائدة و سنكتشفها بسرعة.

3.14 الماكرو (Macro)

كما قد رأينا بانه باستعمال الـ `#define` ، بامكاننا أن نطلب من المعالج القبلي استبدال كلمة بقيمتها في الشفرة بأكملها.
مثال :

```c
#define NUMBER 9
```

و الذي يعني أنّ جميع `NUMBER` في الشفرة يتمّ استبدالها بـ9. لقد رأينا أنّها تعمل كوظيفة بحث و استبدال يقوم بها المعالج القبلي قبل الترجمة.

لديّ خبر جديد ! في الواقع `#define` أقوى من هذا بكثير. فهي قادرة على الاستبدال بـ... شفرة مصدرية بأكملها ! عندما نستخدم `#define` للبحث و استبدال كلمة بشفرة مصدرية نقول أننا أنشأنا ماكرو **(Macro)**.

ماكرو بدون معاملات

هذا مثال عن ماكرو بسيطة :

```c
#define COUCOU() printf("Coucou");
```

الشيء الذي تغيّر هو القوسين الذين أضفناهما بعد الكلمة المفتاحيّة (هنا `COUCOU()`). سنرى فائدتهما بعد قليل.

فلنجرب الماكرو داخل الشفرة المصدرية :

```c
#define COUCOU() printf("Coucou");
int main(int argc, char *argv[])
{
  COUCOU()
  return 0;
}
```

```
Coucou
```

أعلم أنّ هذا ليس شيئا جديدا حاليًا. لكنّ الذي عليك فهمه، هو أن الماكرو عبارة عن بضعة أسطر من الشفرة التي يتم استبدالها مباشرة في الشفرة قبل الترجمة.
الشفرة التي كتبناها تصبح هكذا قبل الترجمة :

```c
int main(int argc, char *argv[])
{
        printf("Coucou");
        return 0;
}
```

إذا فهمت هذا فقد فهمت مبدأ عمل الماكرو.

؟ لكن، هل يمكننا أن نضع سطراً واحدا فقط من الشفرة في كلّ ماكرو ؟

لا، لحسن الحظ يمكنك وضع عدّة أسطر من الشفرة في المرّة. يكفي وضع `\` قبل كلّ سطر جديد، مثل هذا :

```
1  #define TELL_YOUR_STORY() printf("Hello, my name is Brice\n"); \
2                            printf("I live at Nice\n"); \
3                            printf("I love rice\n");
4  int main(int argc, char *argv[])
5  {
6          TELL_YOUR_STORY()
7          return 0;
8  }
```

```
Hello, my name is Brice
I live at Nice
I love rice
```

كما تلاحظ في `main`، أنّ استدعاء الماكرو لا يوضع بعده فاصلة منقوطة في النهاية. في الواقع، لأنها توجيهة خاصة بالمعالج القبلي و لا تحتاج إلى أن تنتهي بفاصلة منقوطة.

ماكرو بالمعاملات

لحدّ الآن، رأينا كيف نقوم بإنشاء ماكرو بدون معاملات، أي بقوسين فارغين. الفائدة من هذا النوع من الماكرو أنّه يفيد في "اختصار" شفرة طويلة، خاصّة إذا كانت ستتكرّ كثيرا في شفرتك المصدريّة.

لكن الماكرو تصبح مفيدة أكثر عندما نضع لها الأقواس. هذا يعمل تقريبا مثل الدوال :

```
1  #define ADULT(age) if (age >= 18) \
2                          printf("You are adult\n");
3  int main(int argc, char *argv[])
4  {
5          ADULT(22)
6          return 0;
7  }
```

```
You are adult
```

م يمكننا مثلا إضافة الـ `else` لكي نُظهر على الشاشة : أنت لست بالغاً "You are not adult". حاول القيام بذلك، الأمر ليس صعبا. لا تنس وضع الشرطة الخلفيّة `\` قبل السطر الجديد.

مبدأ الماكرو بسيط جدّا :

```
#define ADULT(age) if (age >= 18) \
                   printf("You are adult\n");
```

نقوم بوضع اسم "متغير" بين القوسين، و الذي نسميه `age` . في كلّ شفرة الماكرو، `age` سيتم استبداله بالعدد المحدد عند الاستدعاء (هنا 22).

أي أن الشفرة المصدريّة السابقة بعد مرور المعالج القبلي مباشرة تصبح هكذا :

```
int main(int argc, char *argv[])
{
        if (22 >= 18)
                printf("You are adult\n");
        return 0;
}
```

تم استبدال السطر الذي يستدعي الماكرو بالشفرة التي تحتويه الماكرو، و تم تعويض "المتغير" `age` بقيمته مباشرة في الشفرة المصدريّة للاستبدال.

يمكننا إنشاء ماكرو بعدة معاملات :

```
#define ADULT(age, name) if (age >= 18)\
                         printf("You are adult %s\n", name);
int main(int argc, char *argv[])
{
        ADULT(22, "Maxime")
        return 0;
}
```

هذا كلّ ما يمكننا أن نقوله حول الماكرو و المميزات التي تقدّمها لنا. يمكنك تذكّر أنّه مجرّد استبدال للشفرة المصدرية يمكنه استخدام المعاملات.

> في الواقع، أنت لست بحاجة أن تتعامل كثيراً مع الماكرو، لكن اعلم أن مكتبات معقدة كالـwxWidgets و Qt (مكتبات لإنشاء الواجهات الرسوميّة) تستعملان بكثرة الماكرو. لهذا من المستحسن أن تتعلّم كيف تعمل الأمور من الآن كي لا تضيع لاحقاً.

4.14 الشروط

أجل : يمكننا أن نستعمل الشروط في لغة المعالج القبلي ! لاحظ كيف تعمل :

```
#if condition
  /* Code to compile if the condition is true */
#elif condition2
  /* Else, compile this code if the condition2 is true */
#endif
```

الكلمة المفتاحية `#if` تسمح بإدراج شرط معالج قبلي، `#elif` تعني `else if` . الأمر يتوقف عندما نضع `#endif` ، تلاحظ أنه لا توجد حاضنتان في لغة المعالج القبلي.

الفائدة هي أننا سنتمكن من إجراء **ترجمة شرطية (Conditional compilation)**.
في الواقع، إن كان الشرط محققا فإن الشفرة التالية ستتم ترجمتها، و إلّا فسيتم حذفه و لن يكون جزءً من البرنامج النهائي.

#ifdef و #ifndef

سنرى الآن الفائدة من استعمال `#define` لتعريف ثابت دون إعطائه أيّ قيمة، مثلما علّمتك من قبل :

```
#define CONSTANT
```

في الواقع، يمكننا استعمال الشرط `#ifdef` لنقول "إن كان الثابت معرّفا". بالنسبة لـ `#ifndef` ، فهذا يعني "إن كان الثابت غير معرّف".

يمكننا أن نتخيل هذا :

```
#define WINDOWS
#ifdef WINDOWS
  /* Source code for Windows */
#endif
#ifdef LINUX
  /* Source code for Linux */
#endif
#ifdef MAC
  /* Source code for Mac */
#endif
```

هذا مثال عن برنامج متعدد المنصات (multi-platform) للتلاؤم مع النظام مثلا.
إذن، يجب من أجل كلّ نظام إعادة ترجمة الشفرة (هذا ليس أمرا سحريّا). إن كنت في Windows فستكتب `#define WINDOWS` في الأعلى و تعيد الترجمة.
إن أردت الترجمة لـLinux فسيكون عليك تغيير `#define` لوضع `#define LINUX` و تعيد الترجمة. هذه المرّة الجزء الخاصّ بـLinux الّذي ستتمّ ترجمته أمّا باقي الشروط فلن تكون محققة يعني أنه سيتم تجاهلها.

#ifndef لتفادي التضمينات اللامنتهية

`#ifndef` مهمّة جدّا في الملفّات `.h` لتجنّب "التضمينات اللامنتهية".

؟ ماذا يعني التضمين اللامنتهي ؟

هذا أمر بسيط، تخيل أن لدينا ملفاً `A.h` و ملفاً `B.h` ، الملف `A.h` يحتوي `#include` للملف `B.h` . إذا فالملف `B.h` مضمّن الآن بـ `A.h` .

و هنا يبدأ المشكل، تخيّل أن الملف `B.h` يحتوي نفسه على `#include` للملف `A.h` ! هذا يحدث أحيانا في البرمجة ! يعني أن الملف الأول بحاجة إلى الثاني و الثاني بحاجة إلى الأول أيضا.

إن فكّرنا قليلا، فسنعرف أنّ هذا ما سيحصل :

- الحاسوب يقرأ `A.h` و يجد بأن عليه تضمين `B.h`.
- يقوم بقراءة `B.h` فيجد بأن عليه تضمين `A.h`.
- يضمّن `A.h` في `B.h`، لكن داخل `A.h` يجد بأنّه يحتاج إلى تضمين `B.h` !
- يكرّر، يرى أنّ `B.h` و يجد أنّه يجب عليه تضمين `A.h`.
- إلخ.

قد تظنّ أنّ هذا الأمر لا نهاية له !
في الحقيقة، من كثرة التضمينات، سيتوقّف المعالج القبلي قائلا "لقد سئمت من التضمينات !" و هذا ما يعطّل الترجمة.

كيف السبيل لوقف هذا الكابوس المريع ؟ إليك الحيلة. أطلب منك فعل هذا في **كلّ ملفّات `.h` الخاصّة بك** بدون استثناء :

```
1 #ifndef DEF_FILENAME // If the constant has not been defined, the file then has
      never been included
2 #define DEF_FILENAME // We define the constant so the file will not be included
      the next time
3
4 /□ Content of your file .h (other #include, prototypes, #define...) □/
5
6 #endif
```

أي أننا نضع كل محتوى الملف `.h` (بما في ذلك `#include`، النماذج، `#define`) بين الـ `#ifndef` و الـ `#endif`.

هل فهمت كيف تعمل الشفرة ؟ أوّل مرّة رأيت هذه التقنيّة كنت مشوّشا كثيرا : سأحاول أن أشرح.

تخيل أن الملف `.h` يتم تضمينه للمرة الأولى، سيقرأ الحاسوب الشرط "إذا كان الثابت `DEF_FILENAME` لم يتم تعريفه". بما أنه يتمّ قراءة الملف للمرة الأولى، فإن الثابت لم يتم تعريفه بعد، فسيقوم المعالج القبلي بالدخول إلى داخل `if`.

أوّل تعليمة سيجدها هي :

```
1 #define DEF_FILENAME
```

الآن لقد تم تعريف الثابت. في المرّة القادمة التي يتم فيها تضمين الملف، لن يكون الشرط فيها صحيحا و لهذا لن نخاطر بإعادة تضمين الملف من جديد.

يمكنك تسمية اسم الثابت كما تريد، أنا اعتدت على تسميته `DEF_FILENAME`.

الشيء الأهمّ، و الّذي أتمنّى أنّك فهمته جيّدا، هو أن تغيّر اسم الثابت من ملف `.h` إلى آخر. يجب ألّا يكون نفس الثابت في كلّ ملفّات `.h`، و إلا فلن تتم قراءة سوى أول ملف `.h` و الباقية سيتم تجاهلها !
إذا فلتغيّر `FILENAME` إلى اسم الملف الحالي.

> م
> أنصحك بإلقاء نظرة على `.h` الخاصّة بالمكتبات القياسيّة المتواجدة في حاسوبك، سترى بأنها كلها مكتوبة بنفس المبدأ (`#ifndef` في البداية و `#endif` في النهاية). هذا يضمن عدم إمكانيّة حصول تضمينات لا منتهية.

ملخّص

- المعالج القبلي هو برنامج يحلّل الشفرة المصدريّة، و يقوم بإجراء تغييرات عليها قبل الترجمة.
- تعليمة المعالج القبلي `#include` تسمح بإدراج محتوى ملف آخر.
- تعليمة `#define` تسمح بتعريف ثابت معالج قبلي. يتم استبدال كلمة مفتاحيّة بقيمة في الشفرة المصدريّة.
- الماكرو هي مجموعة أسطر من الشفرة الجاهزة معرّفة بالـ`#define`. يمكنها أن تقبل معاملات.
- من الممكن كتابة شروط في لغة المعالج القبلي لاختيار ما يجب ترجمته، نستعمل عادة `#if`، `#elif` و `#endif`.
- لنتجنب أنّ ملفّاً `.h` يتمّ تضمينه عددا لامنتهيا من المرّات، نحميه بمجموعة من ثوابت المعالج القبلي و الشروط. كلّ ملفّاتك `.h` المستقبليّة يجب حمايتها بهذه الطريقة.

الفصل 15

أنشئ أنواع متغيرات خاصّة بك

تسمح لغة الـC بالقيام بشيء يعتبر قوياً جداً : و هو أن ننشئ أنواعاً خاصة بنا، "أنواع متغيّرات مخصّصة". سنرى نوعين : الهياكل (Structures) و التعدادات (Enumerations).

إن إنشاء أنواع خاصّة بنا يعتبر أمراً ضروريا خاصة إذا أردنا إنشاء برامج أكثر تعقيداً.

الأمر ليس (لحسن الحظّ) بالصعب، لكن ركّز جيدّا لأننا سنستعمل الهياكل كل الوقت انطلاقا من الفصل القادم.
يجب أن تعلم أنّ المكتبات تنشئ غالبا أنواعها الخاصّة. لن يمرّ وقت كثير حتّى تستخدم نوعا يدعى "ملف"، و بعده بقليل، أنواع أخرى مثل "نافذة"، "صوت"، "لوحة مفاتيح"، إلخ.

1.15 تعريف هيكل

الهيكل هو تجميع لعدد من المتغيرات التي يمكن لها أن تحمل أنواعا مختلفة. على عكس الجداول التي ترغمنا على استعمال خانات من نفس النوع في كلّ الجدول، بإمكانك تعريف هيكل يحمل الأنواع : `long`، `char`، `int` و `double` في مرّة واحدة.

الهياكل في أغلب الأحيان معرّفة في ملفات `.h` مثلما رأينا مع `#define` و نماذج الدوال. هذا مثال عن هيكل :

```
struct StructureName
{
	int variable1;
	int variable2;
	int anotherVariable;
	double decimalNumber;
};
```

لتعريف هيكل، يجب علينا أن نبدأ بالكلمة المفتاحية `struct`، متبوعة باسم الهيكل (مثلا `File` أو `Screen`).

> شخصيّا لديّ عادة في تسمية هياكلي بنفس قواعد تسمية المتغيّرات، باستثناء أنّي أجعل أوّل حرف كبيرا للتفريق.
> هكذا، عندما أرى الكلمة `captainAge` في شفرتي، أعلم أنّها متغيّر لأنّها تبدأ بحرف صغير. عندما أرى `AudioPart` فأعلم أنّها هيكل (نوع مخصّص) لأنّها تبدأ بحرف كبير.

بعد ذلك، نفتح حاضنة لنغلقها لاحقاً تماما مثل الدوال.

> X
> احذر، الأمر خاصّ هنا : بالنسبة للهياكل، يجب أن تضع بعد الحاضنة النهائية فاصلة منقوطة. هذا أمر إجباري. إن لم تفعله فستتوقّف الترجمة.

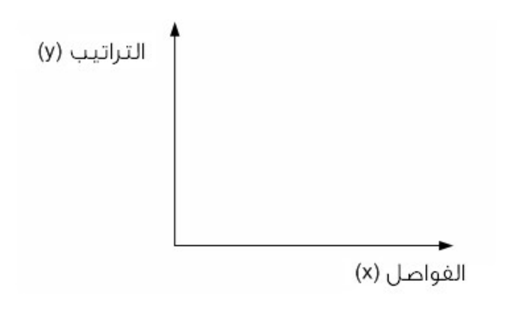
و الآن، ماذا نضع داخل الحاضنتين ؟
هذا سهل، سنضع المتغيرات التي يتكون منها الهيكل، و عادة ما يتكون الهيكل من "مُتَغَيّرَيْن داخِلِيَّيْن" على الأقل، و إلا فلن يحمل معنى كبيرا.

كما ترى، فإنشاء نوع متغيّرات مخصّص ليس بالأمر الصعب. كلّ الهياكل ماهي إلّا "تجميعات" لمتغيّرات من أنواع قاعديّة مثل `long`، `int`، `double`، إلخ. لا توجد معجزة، إنّ نوعا `File` مثلاً ما هو إلا مجموعة من الأعداد القاعديّة !

مثال عن هيكل

تخيل أنك تريد إنشاء متغيّر لكي يُخزّن إحداثيات نقطة في معلم الشاشة. ستحتاج بالتأكيد إلى هيكل كهذا عندما تبدأ في برمجة ألعاب ثنائية الأبعاد في الجزء التالي من الكتاب، هذه إذن فرصة للتقدّم قليلا.

إذا كانت كلمة "علم الهندسة" تُحدث ظهور بقَع غير مفهومة على كامل وجهك، فالمخطّط التالي سيذكّرك قليلا بأساسيّات الأبعاد الثنائيّة (2D).



عندما نعمل في 2D لدينا محوران : محور الفواصل (من اليسار إلى اليمين) و محور التراتيب (من الأسفل إلى الأعلى).
من العادة أن نرمز للفواصل بمتغيّر يدعى `x` و للتراتيب بـ `y`.

هل يمكنك كتابة هيكل `Coordinates` يسمح بتخزين كلّا من الفاصلة (`x`) و الترتيبة (`y`) لنقطة ما ؟
هيّا، هيّا، الأمر ليس صعبا :

```c
struct Coordinates
{
        int x; // Abscissas
        int y; // Ordinates
};
```

هيكلنا يسمّى `Coordinates` و هو متكوّن من متغيرين `x` و `y` أي الفاصلة (Abscissa) و الترتيبة (Ordinate).

إن أردنا، يمكننا بسهولة إنشاء هيكل `Coordinates` من أجل 3D : يكفي فقط إضافة متغيّر ثالث (مثلا `z`) يدلّ على الارتفاع. بهذا سيكون لدينا هيكل لإدارة النقاط الثلاثيّة الأبعاد في الفضاء !

جدول داخل هيكل

يمكن للهياكل أن تحتوي على جداول. هذا جيّد، إذ يمكننا أن نضع داخلها جداول `char` (سلاسل محرفيّة) بدون أيّة مشاكل.
فلنتخيل هيكلاً `Person` و الذي يحتوي على معلومات عن شخص :

```c
struct Person
{
        char firstName[100];
        char lastName[100];
        char address[1000];
        int age;
        int boy; // Boolean : 1 = boy, 0 = girl
};
```

هذا الهيكل متشكّل من 5 متغيرات داخليّة، الثلاث الأولى هي سلاسل محرفيّة لتخزين الاسم، اللقب و العنوان.
المتغيران الأخيران يخزّنان عمر و جنس الشخص. الجنس هو متغيّر منطقي، 1 = صحيح = ولد و 0 = خطأ = بنت.

يمكن لهذا الهيكل أن يساعدنا في كتابة برنامج مذكّرة عناوين. يمكنك بالطبع إضافة القدر الذين تريد من المتغيرات داخل الهيكل من أجل إتمامها إذا أردت. لا يوجد حدّ لعدد المتغيّرات في هيكل.

2.15 استعمال هيكل

و الآن، بما أن الهيكل معرّف في ملف `.h`، سنتمكّن من استعماله في دالة موجودة بملف `.c`.
أنظر كيف نقوم بإنشاء متغير من نوع `Coordinates` (الهيكل الذي عرّفناه سابقا) :

```c
#include "main.h" // Including the files that contains the prototypes and structures
int main(int argc, char *argv[])
{
        struct Coordinates point; // Creating a variable "point" of type Coordinates
        return 0;
}
```

هكذا نكون قد أنشأنا متغيراً `point` من نوع `Coordinates` ! هذا المتغير سيحمل داخله مركّبين (متغيرين داخليين) : `x` و `y` (فاصلته و ترتيبته).

> ؟ هل من اللازم أن نضع الكلمة المفتاحية `struct` عند تعريف المتغير ؟

نعم، فهذا يسمح للحاسوب بأن يفرّق بين نوع عادي (مثل `int`) و نوع مخصّص.
المبرمجون وجدوا أنه من المتعب جدّا أن يكتبوا في كلّ مرة الكلمة `struct` في كلّ تعريف لمتغيّر مخصّص. لمعالجة هذا المشكل، اخترعوا تعليمة خاصّة : الـ `typedef`.

الـtypedef

لنعد إلى الملف `.h` الذي يحمل تعريف هيكلنا من نوع `Coordinates`. سنضيف تعليمة اسمها `typedef` و الّتي تفيد في إعطاء اسم مستعار (alias) لهيكل، أي كتابة شيء مكافئ لكتابة آخر.

إذا، سنضيف سطرا يبدأ بـ `typedef` قبل تعريف الهيكل مباشرة :

```
typedef struct Coordinates Coordinates;
struct Coordinates
{
        int x;
        int y;
};
```

هذا السطر متكون من ثلاثة أجزاء :

- `typedef` : تعني أننا سنقوم بإنشاء اسم مستعار لهيكل.
- `struct Coordinates` : هو اسم الهيكل الذي سنقوم بانشاء اسم مستعار له (أي "مكافئ").
- `Coordinates` : هو الاسم المكافئ.

ببساطة، هذا السطر يقول : "كتابة `Coordinates` مكافئ لكتابة `struct Coordinates`". بفعل هذا، لن يكون عليك كتابة الكلمة `struct` عند كلّ تعريف لمتغيّر من نوع `Coordinates`. يمكننا العودة إلى `main` و كتابة فقط :

```
int main(int argc, char *argv[])
{
        Coordinates point; // The computer understands that we are talking about "struct Coordinates" thanks to typedef
    return 0;
}
```

أنصحك أن تستعمل الـ `typedef` مثلما فعلت أنا هنا من أجل `Coordinates`. أغلب المبرمجين يفعلون هذا. هذا يسمح لهم بعدم كتابة `struct` في كلّ مرّة. المبرمج الجيّد هو مبرمج كسول ! أي أنه يكتب أقل ما يمكن.

تغيير مركّبات هيكل

و الآن بعدما قمنا بإنشاء متغيّرنا `point` ، نريد أن نغيّر إحداثيّاته.
كيف نصل إلى `x` و `y` الموجودة في المتغير `point` ؟ هكذا :

```
int main(int argc, char *argv[])
{
        Coordinates point;
        point.x = 10;
        point.y = 20;
        return 0;
}
```

بهذا نكون قد غيّرنا قيمة `point` ، بإعطائه الفاصلة 10 و الترتيبة 20. نقطتنا أصبحت في الوضعية (10؛20) (هذا هو الترميز الرياضياتي للإحداثيّات).

لكي نتمكن من الوصول إلى مركّب في الهيكل، يجب كتابة :

```
variable.componentName
```

النقطة هي التي تفرّق بين المتغير و المركّب.

إن أخذنا الهيكل `Person` الذي رأيناه منذ قليل و نطلب الاسم و اللقب فسنفعل هكذا :

```
int main(int argc, char *argv[])
{
        Person user;
        printf("What's your last name ? ");
        scanf("%s", user.lastName);
        printf("What's your first name ? ");
        scanf("%s", user.firstName);
        printf("You are %s %s", user.firstName, user.lastName);
        return 0;
}
```

```
What's your last name ? Dupont
What's your first name ? Jean
You are Jean Dupont
```

نرسل المتغير `user.lastName` إلى الدالة `scanf` ، و التي ستكتب مباشرة في `user` .
نفعل نفس الشيء مع `firstName` ، يمكننا فعل ذلك أيضا مع العنوان، العمر و الجنس، لكنّي لا أرغب بتكرار ذلك (يجب أن أكون مبرمجا !).

يمكن فعل هذا بدون معرفة الهياكل، فقط بإنشاء متغيّر `lastName` و آخر `firstName` .
لكن الفائدة هنا هي أنه بهذه الطريقة يمكننا أن ننشئ متغيرا آخر من نوع `Person` و يكون لديه هو أيضا اسمه الخاص، لقبه الخاص، إلخ. يمكننا إذن فعل هذا :

```
1 Person player1, player2;
```

و هكذا نخزّن معلومات كلّ لاعب. كلّ لاعب سيكون لديه اسمه الخاص، لقبه الخاص، إلخ.

يمكننا أن نفعل ما هو أفضل : يمكننا تعريف جدول من `Person` !
القيام بهذا سهل :

```
1 Person players[2];
```

و بعدها يمكننا الوصول إلى لقب اللاعب المتواجد بالخانة الأولى مثلا، هكذا :

```
players[0].lastName
```

الفائدة من استعمال الجدول هنا، هو أنها بامكاننا استعمال حلقة لنقرأ المعلومات الخاصة باللاعب 1 و اللاعب 2 بدون الاضطرار إلى إعادة الشفرة مرّتين. يكفي تصفّح الجدول `players` و طلب كلّ مرّة اللقب، الاسم، العنوان ...

تمرين : قم بتعريف جدول من نوع `Person`، و اقرأ المعلومات الخاصة بكلّ لاعب باستخدام حلقة. إبدأ بجدول ذي خانتين، و إن كان ذلك ممتعا، حاول تكبير العدد لاحقا.
في النهاية، عليك بإظهار المعلومات التي أخذتها من كلّ لاعب.

تهيئة هيكل

بالنسبة للهياكل، مثل كلّ المتغيرات، الجداول و المؤشّرات، فنحن نفضّل أن نعطيها قيما ابتدائية كي نضمن أنّها لن تحوي "قيما عشوائية". في الواقع، أعيد تذكيرك، المتغير الّذي يتمّ إنشائه يأخذ القيمة الموجودة في الذاكرة حيث تمّ وضعه. أحيانا تكون هذه القيمة 0، و أحيانا بقايا برنامج مرّ قبلك، لذلك ستكون قيمته شيئا لا معنى له، مثل -84570.

للتذكير، هكذا نقوم بالتهيئة :

- المتغير : نعطيه القيمة 0 (الحالة الأبسط).
- المؤشّر : نجعل قيمته `NULL`. بالمناسبة فـ`NULL` هي `#define` موجود في مكتبة `stdlib.h` و هي عادة 0، لكنّنا نستمرّ في استخدام `NULL` للمؤشرات لكي نبيّن أنّها مؤشّرات و ليست متغيّرات عاديّة.
- الجدول : نضع كلّ خاناته على القيمة 0.

بالنسبة للهياكل، فالتهيئة شبيهة بتلك الخاصّة بالجدول. في الواقع، يمكننا القيام بها عند التصريح عن المتغيّر :

```
1 Coordinates point = {0, 0};
```

و هذا يعرّف بالترتيب : `point.x = 0` و `point.y = 0` . لنعد إلى الهيكل `Person` (الذي يحتوي سلاسل محرفيّة). يمكننا أن نعطي قيمة ابتدائية للسلاسل بكتابة فقط `""` (لا شيء بين علامتي الاقتباس). لم أعلمك هذا الشيء في الفصل الخاصّ بالسلاسل، لكنّ الوقت ليس متأخّرا على تعلّمها. يمكننا إذن تهيئة على الترتيب `firstName`، `lastName`، `address`، `age` و `boy` هكذا :

```
1 Person user= {"", "", "", 0, 0};
```

رغم ذلك، أنا لا أستخدم هذه التقنيّة كثيرا. أفضّل أن أرسل متغيّري، مثلا `point`، إلى دالّة `initializeCoordinates` تقوم بالتهيئات من أجلي على متغيري.
لفعل هذا، يجب إرسال مؤشّر نحو متغيري. في الواقع، إن أرسلت فقط المتغيّر، سيتم إنشاء نسخة عنه (مثل أيّ متغيّر عادي) و تعديل قيم النسخة لا قيم المتغيّر. راجع الخيط الأحمر من فصل المؤشرات إن نسيت كيف يعمل هذا الأمر.

يجب إذن تعلّم كيفيّة استخدام المؤشرات على الهياكل. الأمور بدأت تصعب قليلا !

3.15 مؤشّر نحو هيكل

المؤشّر على الهيكل يتمّ إنشائه بنفس طريقة إنشاء مؤشّر على `int` أو `double` أو أيّ نوع قاعديّ آخر :

```
1 Coordinates* point = NULL;
```

بهذا نكون قد عرّفنا مؤشّرا نحو `Coordinates` اسمه `point`.
و لأن التذكير لن يضرّ أحدا، أعيد إخبارك بأنه من الممكن وضع النجمة أمام اسم المتغيّر، فهذه الكتابة مكافئة تماما للسابقة :

```
1 Coordinates *point = NULL;
```

أنا أفعل هكذا كثيرا، لأنه لتعريف عدّة مؤشرات على سطر واحد، سيكون علينا وضع النجمة أمام اسم كل واحد منها :

```
1 Coordinates *point1 = NULL, *point2 = NULL;
```

إرسال هيكل إلى دالة

الشيء الذي يهمّنا هنا، هو كيفيّة إرسال مؤشر هيكل إلى دالة كي تقوم هذه الأخيرة بتعديل محتواه.

هذا ما سنقوم به في هذا المثال، سنقوم بإنشاء متغير من نوع `Coordinates` في `main`، و نرسل عنوانه إلى `initializeCoordinates`. دور هذه الدالة هو إعطاء القيمة 0 لعناصر الهيكل.

الدالة `initializeCoordinates` ستأخذ معاملا واحدا : مؤشر نحو هيكل من نوع `Coordinates` (أي `*Coordinates`).

```
int main(int argc, char □argv[])
{
        Coordinates myPoint;
        initializeCoordinates(&myPoint);
         return 0;
}
void initializeCoordinates(Coordinates□ point)
{
        // Initializing each member of the structure here
}
```

متغيري `myPoint` تم إنشاؤه في `main`.
نقوم بإرسال عنوانه إلى الدالة `initializeCoordinates` الّتي تسترجعه على شكل متغيّر يسمّى `point` (كان بإمكاننا تسميته كما شئنا، هذا الأمر ليس له أيّ تأثير).

الآن بما أنّنا داخل الدالة `initializeCoordinates`، سنقوم بتهيئة قيم المتغير `point` واحدة بواحدة.
يجب عدم نسيان وضع النجمة أمام اسم المؤشر للوصول إلى المتغير. إن لم تفعل، فأنت تخاطر بتغيير العنوان، و ليس هذا ما نريد فعله.

و لكن هاهي مشكلة ... لا يمكننا القيام مباشرة بهذا :

```
void initializeCoordinates(Coordinates□ point)
{
        □point.x = 0;
        □point.y = 0;
}
```

سيكون ذلك سهلا جدّا ... لماذا لا يمكننا القيام بهذا ؟ لأنّ النقطة تطبّق على `point` فقط و ليس على `*point`.
لكنّ ما نريده هو الوصول إلى `*point` لتغيير قيمته.
لحلّ هذا المشكل، يجب وضع الأقواس حول `*point`، هكذا ستطبّق النقطة على `*point` و ليس فقط على `point` :

```
void initializeCoordinates(Coordinates□ point)
{
        (□point).x = 0;
        (□point).y = 0;
}
```

هذه الشفرة تعمل، يمكنك تجريبها. المتغير من نوع `Coordinates` تمّ إرساله إلى الدالة التي هيّأت `x` و `y` على 0.

> في لغة الـC، نهيّئ عادة هياكلنا بالطريقة الّتي رأيناها سابقا. بالمقابل، في لغة الـ++C، التهيئة تكون في الغالب داخل "دوال". إن لغة ++C ليست سوى "تحسين خارق" للهياكل. كثير من الأشياء تبدأ من هذا و أحتاج إلى كتاب كامل لأتحدّث عنها (كلّ شيء في وقته).

اختصار عمليّ و مستعمل بكثرة

سترى أننا سنستعمل كثيراً مؤشرات نحو هياكل. بصراحة، يجب أن أعترف لك بأنّه في لغة الـC نستخدم المؤشرات نحو الهياكل أكثر من استعمال الهياكل وحدها. لهذا فعندما أقول لك بأنّ المؤشرات ستظلّ تتبعك حتّى إلى قبرك، فأنا لا أقولها تقريبا من أجل المزاح !

بما أن المؤشرات نحو الهياكل مستعملة بكثرة، نجد أنفسنا نستعمل هذه الكتابة كثيرا :

```
(*point).x = 0;
```

مرّة أخرى، المبرمجون وجدوا هذه الكتابة طويلة جدّا. الأقواس حول `*point`، يا لها من بلوى ! إذن، بما أن المبرمجين أشخاص كسالى (لقد قلت هذا سابقا على ما أعتقد)، فقد اخترعوا هذا الاختصار :

```
point->x = 0;
```

هذا الاختصار يتمّ كتابته بمطّة `-` متبوعة بعلامة ترتيب `>`.

كتابة `point->x` هو إذن مكافئ تماما لكتابة `(*point).x`.

> !
> لا تنس أننا لا نستطيع استعمال السهم إلا مع المؤشرات.
> إن كنت تعمل على المتغير مباشرة، يجب عليك استخدام النقطة كما رأينا في البداية.

لنعد إلى دالتنا `initializeCoordinates` يمكننا كتابتها بالشكل التالي :

```
void initializingCoordinates(Coordinates* point)
{
        point->x = 0;
        point->y = 0;
}
```

و تذكّر جيّدا اختصار السهم، سنستعمله كثيراً من الآن. و خاصّة لا تخلط بين السهم و "النقطة". السهم مخصّص للمؤشّرات، "النقطة" مخصّصة للمتغيّرات. استخدم هذا المثال الصغير للتذكّر :

```
int main(int argc, char *argv[])
{
        Coordinates  myPoint;
        Coordinates *pointer = &myPoint;
        myPoint.x = 10; // We work on a variable, we use the "dot"
        pointer->x = 10; // We work on a pointer, we use the arrow
        return 0;
}
```

نغيّر قيمة `x` إلى 10 بطريقتين مختلفتين، هنا : الطريقة الأولى هي بالعمل مباشرة على المتغير، و الطريقة الثانية باستعمال المؤشر.

4.15 التعدادات

التعدادات هي طريقة مختلفة قليلاً لنعرّف نوع متغيرات خاص بنا.

التعداد ليس متكوّنا من "مركّبات" كما هو الحال مع الهياكل. و إنما هو مجموعة من "القيم الممكنة" لمتغيّر. التعداد سيأخذ إذن خانة واحدة في الذاكرة و هذه الخانة تأخذ قيمة واحدة من مجموع القيم التي قمت بتعريفها (واحدة فقط في كلّ مرّة). هذا مثال عن تعداد :

```
typedef enum Volume Volume;
enum Volume
{
        LOW, MEDIUM, HIGH
};
```

تلاحظ أننا نستعمل `typedef` هنا أيضا، مثلما رأينا لحد الآن.

لكي نقوم بتعريف تعداد نستعمل الكلمة المفتاحية `enum`. اسم التعداد هنا هو `Volume`. إنّه نوع مخصّص قمنا بتعريفه يمكن له أن يأخذ واحدة من الثلاث قيم الّتي وضعناها : إما `LOW` أو `MEDIUM` أو `HIGH`.

يمكننا إذن أن نعرّف متغيرا اسما `music` من نوع `Volume` لتخزين مستوى صوت الموسيقى.
يمكننا تهيئة الموسيقى على المستوى `MEDIUM` :

```
Volume music = MEDIUM;
```

يمكننا لاحقاً في الشفرة، أن نغيّر قيمة مستوى الصوت و وضعها إمّا على `HIGH` أو على `LOW`.

إرفاق قيم التعداد بأعداد

قد لاحظت أنّي كتبت القيم الممكنة بأحرف كبيرة. هذا يفترض به أن يذكّرك بالثوابت و `#define`، أليس كذلك ؟

في الواقع، إنّ هذا مشابه كثيرا و لكنّه ليس نفس الشيء.
المترجم يقوم تلقائيا بارفاق قيم التعداد بأعداد موافقة لها.

في حالة تعدادنا `Volume`، `LOW` سيتم ارفاقها بالقيمة 0، `MEDIUM` بالقيمة 1، و `HIGH` بالقيمة 2.
الإرفاق يتمّ تلقائيّا انطلاقا من 0.

خلافا لـ `#define`، فالمترجم هو من يرفق `MEDIUM` بـ1 مثلا، وليس المعالج القبلي. لكنّ هذا سيكون تقريبا مكافئا له.
بطبيعة الحال، عندما هيّئنا المتغيّر `music` على `MEDIUM`، فإنّنا قد وضعنا القيمة 1 في خانة الذاكرة الموافقة.

> ؟ عمليّا، هل يهمّنا أن نعرف أنّ `MEDIUM` تساوي 1، `HIGH` تساوي 2، إلخ. ؟

لا. فهذا حقيقة لا يعنينا. المترجم هو من سيقوم تلقائيّا بإرفاق العدد المناسب إلى كلّ قيمة. بفضل هذا، ليس عليك سوى كتابة :

```
if (music == MEDIUM)
{
	// Play music with medium volume
}
```

لا يهمّ ما هي قيمة `MEDIUM` ، ستترك المترجم يهتمّ بالأعداد.

الفائدة من كلّ هذا ؟ هي أنها تجعل الشفرة قابلة للقراءة جيّدا. في الواقع، أيّ شخص يمكنه بسهولة قراءة `if` السابق نفهم جيّدا أنّ الشرط يعني "إن كانت الموسيقى بمستوى صوت متوسّط").

ارفاق قيمة محددة

حاليّا، كان المترجم هو من يقرّر إرفاق العدد 0 ثم 1، 2، 3 بالترتيب.
من الممكن طلب إرفاق قيمة محدّدة لكلّ عنصر من التعداد.

ما الفائدة التي يمكن تحصيلها من هذا ؟ حسنا فلنفرض أنّه في حاسوبك، الصوت يتم تحديده بقيمة بين 0 و 100 (0 = لا صوت، 100 = 100% من الصوت)، فسيكون من الجيد ارفاق قيمة محدّدة بكلّ عنصر :

```
typedef enum Volume Volume;
enum Volume
{
	LOW = 10, MEDIUM = 50, HIGH = 100
};
```

هنا، المستوى `LOW` يوافق 10% من المستوى، المستوى `MEDIUM` يوافق 50%، إلخ.
يمكننا بسهولة إضافة بعض القيم الأخرى مثل `MUTE` . نرفق في هذه الحالة `MUTE` بالقيمة ... 0 ! لقد فهمت.

ملخّص

- الهيكل هو نوع بيانات مخصّص يمكنك إنشاؤه و استخدامه في برامجك. يجب عليك تعريفه، عكس الأنواع القاعديّة مثل `int` و `double` الّتي نجدها في كلّ البرامج.
- الهيكل يتكوّن من "متغيّرات داخليّة" تكون عادة من أنواع قاعديّة مثل `int` و `double` ، و أيضا من الجداول.
- نستطيع الوصول إلى أحد مركّبات الهيكل بفصل اسم المتغيّر و المركّب بنقطة : `player.firstName` .
- إذا كنّا نتعامل مع مؤشر نحو هيكل و أردنا الوصول إلى أحد مركّباته، نستخدم السهم بدل النقطة : `playerPointer->firstName` .
- التعداد هو نوع مخصّص يمكنه فقط أخذ إحدى القيم المسبقة التعريف : `LOW` ، `MEDIUM` أو `HIGH` مثلا.

الفصل 16

قراءة و كتابة الملفات

المشكل مع استعمال المتغيّرات، هو أنها موجودة فقط في الذاكرة العشوائية RAM. بخروجنا من البرنامج، كلّ المتغيّرات يتم حذفها من الذاكرة و لن يصبح ممكنا استعادة قيمها. كيف يمكننا إذن أن نحتفظ بأحسن العلامات التي تحصّلنا عليها في لعبة ؟ كيف يمكننا إنشاء محرر نصوص إذا كان كلّ النصّ المكتوب يختفي بمجرّد إيقاف البرنامج ؟

لحسن الحظّ يمكننا القراءة من الملفّات و كذا الكتابة فيها في لغة C. هذه الملفّات مُخزّنة في القرص الصلب (Hard disk) الخاص بالحاسوب : الشيء الإيجابيّ إذن هو أنها تبقى محفوظة، حتّى عند إيقاف البرنامج أو الحاسوب.

للقراءة من الملفات و الكتابة فيها، سنحتاج إلى استعمال كلّ ما درسناه حتّى الآن : المؤشرات، الهياكل، السلاسل المحرفيّة، إلخ.

1.16 فتح و غلق ملف

للقراءة و الكتابة في الملفّات، سنستعمل دوالاً معرّفة في المكتبة `stdio` التي استعملناها سابقاً.
نعم، هذه المكتبة تحتوي على الدالتين `scanf` و `printf` اللتان نعرفهما جيّداً ! لكن ليس هذا فحسب : يوجد بها الكثير من الدوال الأخرى، خصوصا التي تعمل على الملفات.

> كل المكتبات التي استعملناها حتّى الآن (`stdlib.h`، `stdio.h`، `math.h`، `string.h` ...) تشكّل ما نسميه بالمكتبات القياسية (Standard libraries)، و هي مكتبات تأتي تلقائيا مع البيئة التطويرية التي تستخدمها و لديها الميزة في أنّها تعمل على كل أنظمة التشغيل. بالإمكان استعمالها في أيّ مكان، سواء كنت في Windows، أو GNU/Linux أو Mac أو غير ذلك. المكتبات القياسيّة ليست كثيرة و لا تمكّننا من القيام بأكثر من بعض الأمور الأساسيّة، كما فعلنا لغاية الآن. للحصول على وظائف أكثر تقدّما، كفتح النوافذ، يجب تنزيل و تثبيت مكتبات جديدة. سنرى ذلك قريبا !

تأكّد إذن، للبدأ، أن تقوم بتضمين المكتبتين `stdio.h` و `stdlib.h` على الأقل أعلى ملفك `.c` :

```
1 #include <stdlib.h>
2 #include <stdio.h>
```

هاتان المكتبتان ضروريتان و أساسيّتان لدرجة أنّي أنصحك بتضمينهما في كلّ البرامج التي تكتبها في المستقبل، أيّا كانت.

حسناً و بعدما قنا بتضمين المكتبتين، يمكننا أن ننطلق في بالأمور الجدّية. إليك الخطوات التي يجب اتّباعها دائماً حينما تريد العمل على ملف، سواء للقراءة منه أو للكتابة فيه :

- نقوم باستدعاء دالة **فتح الملف** `fopen` التي تقوم بإرجاع مؤشّر نحو هذا الملف.
- **نتأكّد من نجاح عمليّة الفتح** (أي إن كان الملفّ موجودا) باختبار قيمة المؤشر الذي أرجعته الدالة. فإن كان المؤشر يساوي `NULL` ، فهذا يعني أنّ فتح الملف لم ينجح، في هذه الحالة لا يمكننا الإكال (يجب أن نظهر رسالة خطأ).
- إذا تم الفتح بنجاح (أي أن قيمة المؤشر تختلف عن `NULL`)، سنستمتع **بالكتابة على الملف أو القراءة منه**، و ذلك باستخدام دوال سنراها لاحقاً.
- بمجرّد أن **ننهي العمل على الملف**، يجب تذكّر "غلقه" باستعمال الدالة `fclose`.

سنتعلّم كخطوة أولى كيف نستخدم `fopen` و `fclose`، حينما نتعلّم هذا، سنتعلّم كيف نقرأ محتواه و نكتب نصّا فيه.

`fopen` : فتح ملف

في فصل السلاسل المحرفيّة، كنا نستعين بنماذج الدوال مثل "دليل استخدام". هذا ما يفعله المبرمجون غالبا : يقرؤون نموذج دالة و يفهمون كيف يستخدمونها. مع ذلك، أعلم أنّنا بحاجة إلى بعض الشروحات البسيطة !

لهذا فلنرى قليلاً نموذج `fopen` :

```c
FILE* fopen(const char* fileName, const char* openMode);
```

هذه الدالة تنتظر معاملين :

- اسم الملف الذي نريد فتحه.
- وضع فتح الملف، أي دلالة تذكر ما الّذي تريد فعله : القراءة من الملف، أو الكتابة فيه، أو كليهما.

هذه الدالة ترجع ... مؤشّرا على `FILE` ! إنّه مؤشّر على هيكل من نوع `FILE`. هذا الهيكل متواجد في المكتبة `stdio.h`. يمكنك فتح الملف لترى مما يتكوّن النوع `FILE`، لكن هذا ليس ما يهمّنا.

؟ لكن لِمَا اسم الهيكل كله بحروف كبيرة ؟ اعتقدت أن الأسماء بالحروف الكبيرة حجزناها للثوابت و لـ `#define` ؟

هذه "القاعدة"، أنا من قت بتحديدها (و كثير من المبرمجين يتبعونها)، و لكنّها لم تكن أبدا مفروضة. و يبدو أنّ من برمجوا `stdio.h` لا يتبعون نفس القواعد !

هذا لا يجب أن يشوّشك كثيرا. سوف ترى أنّ المكتبات الّتي سندرسها لاحقا تتّبع نفس القواعد التي أتّبعها، أي أن اسم الهيكل يبتدئ فقط بحرف واحد كبير.

لنعد إلى دالتنا `fopen`، إنها تقوم بارجاع `FILE*`. إنه من المهم جدّا استرجاع هذا المؤشّر كي نتمكّن لاحقاً من القراءة و الكتابة في الملف. و لهذا سنقوم بإنشاء مؤشّر على `FILE`، في بداية دالتنا (`main` مثلا) :

```
int main(int argc, char *argv[])
{
        FILE* file = NULL;
        return 0;
}
```

لقد هيّأنا المؤشّر على `NULL` من البداية. أذكّرك بأنّ هذه قاعدة أساسيّة أن تهيّأ كلّ المؤشّرات على `NULL` إنّ لم تكن لديك قيمة أخرى لإعطائها. إن لم تفعل ذلك، فأنت تزيد كثيرا خطر وجود أخطاء لاحقا.

> م
>
> إنه ليس ضرورياً أن تكتب `struct FILE* file = NULL`، لأن منشئ `stdio.h` قد وضعوا `typedef` كما علّمتك منذ مدّة قصيرة. لاحظ أن شكل الهيكل قد يتغيّر من نظام تشغيل إلى آخر (لا تملك بالضرورة نفس المركّبات في كل الأنظمة). لهذا فلن نعدّل محتوى `FILE` مباشرة (لا نقوم بـ `file.element` مثلا). بل سنكتفي باستدعاء دوال، تتعامل مع `FILE` نيابة عناً.

الآن سنقوم باستدعاء الدالة `fopen`، و استرجاع القيمة الّتي تعيدها في المؤشر `file`. و لكن قبل هذا يجب أن أشرح لك كيف تستخدم المعامل الثاني `openMode`. في الواقع، هناك شفرة تدلّ للحاسوب على أنك تريد أن تفتح الملف بوضع القراءة فقط، الكتابة فقط أو الاثنين معاً.
هذه هي أوضاع فتح الملف المختلفة :

- "r" : **قراءة فقط (Read only)**. يمكنك قراءة محتوى الملف، و لكن لا يمكنك الكتابة فيه. يجب أن يكون الملف موجوداً من قبل.
- "w" : **كتابة فقط (Write only)**. يمكنك الكتابة في الملف، لكن لا يمكنك قراءة محتواه. إذا لم يكن الملف موجوداً من قبل، فإنه سيتم إنشاؤه.
- "a" : **إلحاق (Append)**. يمكنك الكتابة في الملف، إنطلاقا من نهايته. إن لم يكن الملف موجوداً، فسيتم إنشاؤه.
- "r+" : **قراءة و كتابة (Read and Write)**. يمكنك القراءة من الملف و الكتابة فيه. يجب أن يكون الملف موجوداً من قبل.
- "w+" : **قراءة و كتابة مع مسح المحتوى أوّلا**. سيتم تفريغ الملف من محتواه أولاً، ثم بإمكانك الكتابة فيه و قراءة محتواه بعد ذلك. إن لم يكن الملف موجوداً من قبل، سيتم إنشاؤه.
- "a+" **إلحاق مع القراءة / الكتابة في آخر الملف**. يمكنك القراءة و الكتابة إنطلاقا من نهاية الملف. إن لم يكن موجوداً، سيتم إنشاؤه.

لمعلوماتك، أنا عرضت لك بعضا من أوضاع فتح ملف. في الحقيقة، يوجد ضعفها ! من أجل كل وضع رأيناه هنا، إن أضفت `"b"` بعد المحرف الأول (`"rb"`، `"wb"`، `"ab"`، `"rb+"`، `"wb+"`، `"ab+"`)، فإن الملف سيتم فتحه بالوضع الثنائي (Binary). هذا وضع خاص قليلاً فلن ندرسه هنا. في الواقع وضع النص يختصّ بتخزين ... النص، تماما كما يوحي الاسم (فقط المحارف القابلة للعرض). أما الوضع الثنائي، يسمح بتخزين المعلومات بايتا بايتا (Byte by byte) (أرقام بشكل أساسي). هذا مختلف كثيرا. على أي حال فطريقة العمل هي تقريبا نفس الّتي سنراها هنا.

شخصياً، أستعمل كثيراً الأوضاع : `"r"` (قراءة)، `"w"` (كتابة)، `"r+"` (قراءة و كتابة في آن واحد). وضع `"w+"` خطر قليلاً لأنه يقوم بمسح محتوى الملف مباشرة، بدون أن يطلب التأكيد قبل القيام بذلك. إن هذا الوضع ليس مفيداً إلا إذا أردنا أن نعيد تهيئة الملف أوّلاً. وضع الإلحاق (`"a"`) يمكنه أن يفيد في بعض الحالات، إذا كنت تريد إضافة معلومات إلى نهاية الملف.

> إن كنت تريد قراءة ملفّ، فمن المستحسن وضع `"r"`. بالطبع، الوضع `"r+"` يعمل أيضا، لكن بوضع `"r"` فأنت تضمن أنّ الملفّ لا يمكن تعديله، هذا نوع من الحماية.

إن كتبت دالّةً `loadLevel` (لتحميل مستوى في لعبة مثلا)، الوضع `"r"` كافٍ، أما إن أردت أن تكتب دالةً `saveLevel` (لحفظ المستوى) فستستعمل الوضع `"w"`.

الشفرة التالية ستفتح الملف `test.txt` في وضع `"r+"` (قراءة و كتابة) :

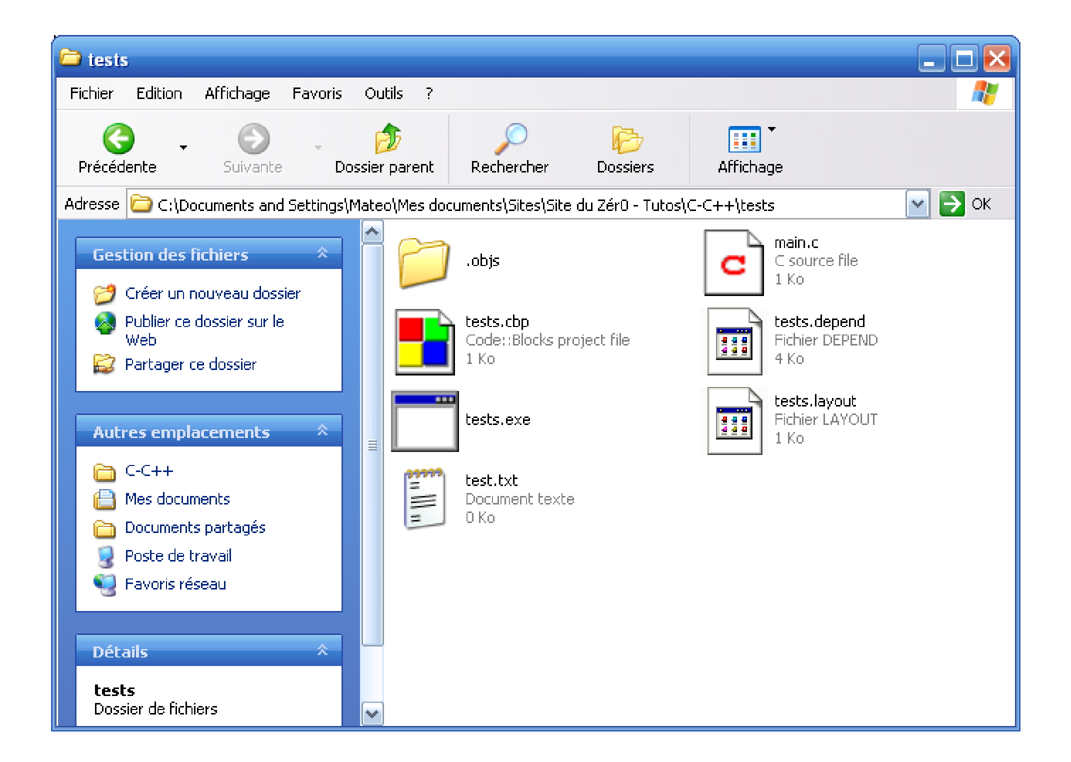
```c
int main(int argc, char *argv[])
{
        FILE* file = NULL;
        file = fopen("test.txt", "r+");
        return 0;
}
```

المؤشّر `file` يصبح إذن مؤشراً على الملف `test.txt`.

> أين يجب أن يكون الملف `test.txt` ؟

يجب أن يكون في نفس المجلّد الذي يتواجد به الملف التنفيذي (`.exe`).

من أجل متطلّبات هذا الفصل، أطلب منك أن تقوم بإنشاء ملف `test.txt` في نفس المسار الذي به `.exe`، مثلما أفعل أنا (الشكل الموالي).



كما ترى فأنا أستعمل حاليّاً بيئة التطوير Code::Blocks الأمر الذي يفسّر وجود ملف المشروع بصيغة `.cbp` (في مكان الصيغة `.sln` إن كنت تستعمل Visual C++ مثلاً). باختصار، الأمر المهم هو أن برنامجي (`tests.exe`) موجود في نفس مجلّد الملف الذي نريد قراءته أو كتابته (`test.txt`).

هل يجب أن يكون الملف بصيغة `.txt` ؟

لا، الأمر يعود إليك في اختيار صيغة الملف عندما تفتحه. أي أنه بإمكانك أن تخترع صيغتك الخاصّة `.level` لحفظ مستويات ألعابك مثلاً.

هل من الواجب أن يكون الملف الذي نريد فتحه في نفس دليل الملف التنفيذي ؟

لا أيضا. يمكنه أن يكون داخل مجلّد بذات الدليل :

```
1 file = fopen("directory/test.txt", "r+");
```

هنا، الملف `test.txt` في مجلّد داخليّ اسمه `directory`. هذه الطريقة التي نسميها المسار النسبي عمليّة أكثر. هكذا، يمكن للبرنامج أن يعمل أينما كان مثبّتا.

من الممكن أيضا فتح ملفّ أينما كان في القرص الصلب. في هذه الحالة يجب كتابة المسار الكامل (ما نسميه المسار المطلق) :

```
1 file = fopen("C:\\Program Files\\Notepad++\\readme.txt", "r+");
```

هذه الشفرة تفتح الملف `readme.txt` الموجود بـ `C:\Program Files\Notepad++`.

> ! تعمّدت استعمال شرطتين خلفيّتين `\` كما تلاحظ. في الواقع، إن كتبت اشارة واحدة، سيعتقد الحاسوب أنني أريد أن استخدم رمزا خاصا (مثل الـ `\n` أو الـ `\t`). لكتابة شرطة خلفيّة في سلسلة، يجب كتابتها إذن مرّتين ! هكذا يمكن أن يفهم أنّك تريد استخدام الرمز `\`.

المشكل مع المسارات المطلقة، هو أنها لا تعمل إلا مع نظام معيّن، فهي ليست حلّا محولا إذن. أي أنه لو كنت تعمل على GNU/Linux لكان عليك كتابة مسار كهذا مثلا :

```
1  file = fopen("/home/mateo/directory/readme.txt", "r+");
```

لهذا فأنا أنصحك بكتابة مسارات نسبية. لا تستعمل المسارات المطلقة إلا في حالة كان البرنامج مخصص لنظام تشغيل معيّن، ليعدّل على ملف معيّن في القرص الصلب.

اختبار فتح ملف

المؤشّر `file` يجب أن يحوي عنوان الهيكل من نوع `FILE`، و الذي نستعمله كواصف (Descriptor) للملف. هذا الواصف تم تحميله من أجلك في الذاكرة من طرف الدالة `fopen`. بعد هذا، هناك احتمالان :

- إمّا أن تنجح عملية الفتح، فستتمكن من المواصلة (أي البدء في القراءة و الكتابة في الملف).
- إمّا ألّا تنجح لأن الملف ليس موجوداً أو أنه مستخدم من طرف برنامج آخر. في هذه الحالة، ستتوقف عن العمل على الملف.

مباشرة بعد فتح الملف، يجب التأكد ما إن تمت العملية بنجاح، أم لا. هذا أمر بسيط : إذا كانت قيمة المؤشر تساوي `NULL`، فإن الفتح قد فشل. إن كانت قيمته تساوي شيئا غير `NULL`، فقد تم الفتح بنجاح. سنتبع إذن هذا المخطط التالي :

```
1  int main(int argc, char *argv[])
2  {
3          FILE* file = NULL;
4          file = fopen("test.txt", "r+");
5          if (file != NULL)
6          {
7                  // We can read or write in the file
8          }
9          else
10         {
11                 // We display an error message if we want
12                 printf("Can't open the file test.txt");
13         }
14         return 0;
15 }
```

افعل هذا دائما عند فتح أي ملف. إن لم تفعل و الملف غير موجود، فأنت تخاطر بتوقّف البرنامج بعدها.

غلق الملف : fclose

إذا نجحت عملية فتح الملف، يمكننا القراءة و الكتابة فيه (سنرى كيف نفعل هذا لاحقاً).
ما إن نكمل العمل على الملف، يجب علينا "غلقه". نستعمل من أجل هذا الدالة `fclose` التي تقوم بتحرير الذاكرة. يعني أنّه سيتم حذف الملف المحمّل في الذاكرة العشوائية.

نموذج الدالة :

```c
int fclose(FILE* pointerOnFile);
```

هذه الدالة تأخذ معاملا واحدا : المؤشر نحو الملف.

تقوم بإرجاع `int`، و الذي يأخذ القيم :

- `0` : إذا نجح غلق الملف.
- `EOF` : إذا فشل الغلق. `EOF` هي عبارة عن `#define` موجودة في `stdio.h` و هي توافق عدداً خاصاً، يُستعمل للقول أنه حصل خطأ، أو أننا وصلنا إلى نهاية الملف. في حالتنا هذه، هذا يعني حدوث خطأ.

في غالب الأحيان، تنجح عملية غلق الملف : هذا ما يدفعني إلى عدم اختبار إن كانت `fclose` قد عملت. رغم هذا، يمكنك فعل ذلك إن أردت.

لإغلاق الملف، نكتب إذن :

```c
fclose(file);
```

في النهاية، المخطّط الذي نتّبعه لفتح و غلق ملف سيكون كالتالي :

```c
int main(int argc, char *argv[])
{
        FILE* file = NULL;
        file = fopen("test.txt", "r+");
        if (file != NULL)
        {
                // We read and we write in the file
                // ...
                fclose(file); // We close the opened file
        }
        return 0;
}
```

لم أستعمل `else` لأظهر رسالة خطأ في حال لم ينجح الفتح، يمكنك فعل ذلك إن أردت.

يجب دائما التفكير في غلق الملف الذي فتحته بمجرّد الإنتهاء من العمل عليه. هذا سيسمح بتحرير الذاكرة.
إن نسيت تحرير الذاكرة، قد يأخذ برنامجك جما كبيراً من الذاكرة بدون أن يستخدمه. في مثال صغير كهذا الأمر غير خطير، لكن مع برنامج كبير، مرحباً بالمشاكل !

نسيان تحرير الذاكرة أمر يقع. بل سيحدث لك هذا كثيرا. في هذه الحالة نقول أنّه قد حدث تسريب للذاكرة (Memory leak). هذا يجعل برنامجك يستخدم قدرا من الذاكرة أكبر من اللازم بدون أن تفهم سبب حصول ذلك. في غالب الأحيان، يكون السبب واحدا أو إثنين من الأمور "الثانوية" مثل نسيان `fclose`.

2.16 طرق مختلفة للقراءة و الكتابة في الملفات

و الآن مادمنا تعلّمنا كيف نفتح و نغلق ملفا، لم يبق سوى أن نضيف الشفرة الّتي تقوم بالقراءة و الكتابة عليه.

سنبدأ برؤية كيفيّة الكتابة في ملفّ (الأمر الأبسط قليلا)، ثمّ نمرّ بعدها إلى كيفيّة القراءة من ملفّ.

الكتابة في ملف

توجد الكثير من الدوال التي تسمح بالكتابة في ملف. يبقى عليك أن تختار أيها الأنسب لك لتستخدمها. هذه الثلاث دوال الّتي سنتعلّمها :

- `fputc` : تكتب حرفا في الملفّ (حرف واحد في المرة).
- `fputs` : تكتب سلسلة محرفيّة في الملف.
- `fprintf` : تكتب سلسلة "منسّقةً" في الملف، طريقة عملها مطابقة تقريبا للدالة `printf`.

fputc

هذه الدالة تكتب حرفا واحدا في المرّة في الملف. نموذجها :

```c
int fputc(int character, FILE* pointerOnFile);
```

و هي تأخذ معاملين :

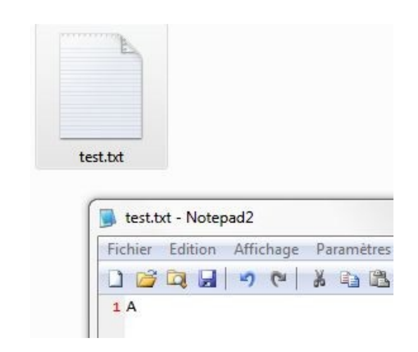
- الحرف الذي يجب كتابته (من نوع `int`، مثلما قلت فاستعماله يعود تقريباً إلى استعمال `char`، إلا أن عدد المحارف الممكن استعمالها هنا أكبر). يمكنك إذن أن تكتب مباشرة `'A'` كمثال.
- المؤشّر نحو الملف الذي نريد أن نكتب فيه. في مثالنا، المؤشّر اسمه `file`. استعمال المؤشّر في كلّ مرة يساعدنا لأنه بإمكاننا أن نفتح العديد من الملفات في آن واحد، و نقرأ و نكتب في كلّ واحد من هذه الملفّات. لست محدّدا بفتح ملفّ واحد في المرّة.

الدالة تقوم بإرجاع `int`، و هو رمز الخطأ. هذا الـ`int` يساوي `EOF` إذا فشلت الكتابة، و إلّا فسيأخذ قيمة أخرى.
بما أنّ الملفّ قد تمّ فتحه بنجاح، فليس من عادتي إختبار إن كانت كلّ واحدة من `fputc` قد نجحت، ولكن يمكنك فعل ذلك إن أردت.

الشفرة التالية تسمح بكتابة الحرف `'A'` في الملف `test.txt` (إذا كان موجوداً من قبل فإنه سيتم استبداله، أمّا إن لم يكن موجوداً سيتم إنشاؤه). الشفرة تحتوي كل الخطوات التي تكلّمنا عنها سابقاً : فتح الملف، اختبار الفتح، الكتابة و الغلق :

```c
int main(int argc, char *argv[])
{
    FILE* file = NULL;
    file = fopen("test.txt", "w");
    if (file != NULL)
    {
        fputc('A', file); // Write the character A
        fclose(file);
    }
    return 0;
}
```

افتح بنفسك الملف `test.txt` . ماذا ترى ؟
إن هذا سحريّ، الملف يحتوي الآن على الحرف `'A'` كما ترى في الشكل التالي :



fputs

هذه الدالة شبيهة جداً بالدالة `fputc` ، إلا أنها تسمح بكتابة سلسلة محرفيّة كاملة، و هذا عادة أحسن من الكتابة حرفاً حرفاً.
لكن `fputc` تبقى ضروريّة حينما نحتاج إلى الكتابة محرفا بحرف، و هذا يحدث كثيرا.

نموذج الدالة :

```c
char* fputs(const char* string, FILE* pointerOnFile);
```

المعاملان سهلا الفهم :

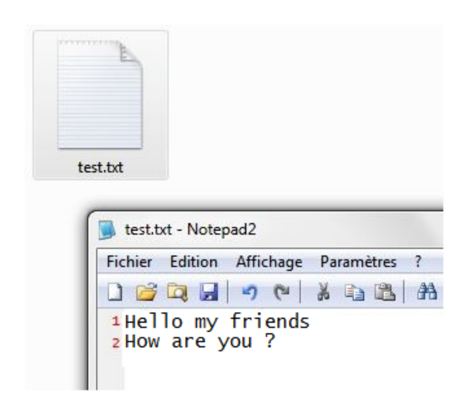
- `string` : السلسلة الّتي نريد كتابتها. تلاحظ أن النوع هنا هو `const char*` : إضافة الكلمة `const` في النموذج تشير إلى أن السلسة الّتي سنعطيها للدالة تُفترض ثابتة. أي أنّ الدالة لن تقوم بتغييرها. هذا أمر منطقي عندما نفكّر فيه : `fputs` يجب أن تقرأ السلسلة بدون تعديلها. هذه إذن معلومة لك (و حماية) أنّ سلسلتك لن يتمّ إدخال أيّة تعديلات عليها.
- `pointerOnFile` : مثل `fputc` ، تحتاج هذه الدالة إلى مؤشر من نوع `FILE*` نحو الملف الذّي فتحته.

الدالّة تعيد القيمة ، `EOF` في حالة وجود خطأ، و إلّا، فهذا يعني أنّها عملت على ما يرام. و هنا أيضا، لن أقوم عادة باختبار القيمة التي ترجعها الدالة.

فلنجرب كتابة سلسلة في ملف :

```
int main(int argc, char *argv[])
{
        FILE* file = NULL;
        file = fopen("test.txt", "w");
        if (file != NULL)
        {
                fputs("Hello my friends\nHow are you ?", file);
                fclose(file);
        }
        return 0;
}
```

الشكل التالي يظهر الملف بعد التعديل عليه من طرف البرنامج :



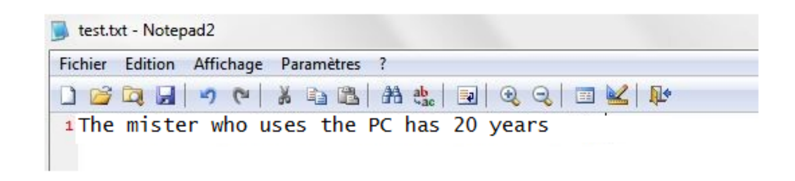
fprintf

إليك نوعاً آخراً من الدالة `printf` . هذه تستخدم للكتابة في ملف. هذه الدالة تستعمل بنفس الطريقة التي نستعمل بها `printf` ، إلا أنه يجب إعطاؤها المؤشر نحو `FILE` كعامل أوّل.

الشفرة التالية تطلب من المستخدم إدخال عمره، ثمّ تقوم بكتابته في الملف :

```
int main(int argc, char *argv[])
{
        FILE* file = NULL;
        int age = 0;
        file = fopen("test.txt", "w");
        if (file != NULL)
        {
                // Request the age
                printf("How old are you ? ");
                scanf("%d", &age);
```

```c
11              // Write the age on the file
12              fprintf(file , "The mister who uses the PC has %d years", age);
13              fclose(file);
14          }
15          return 0;
16  }
```



يمكنك إذا إعادة استعمال ما تعرفه عن `printf` للكتابة في ملف ! لهذا السبب أنا غالبا ما استعمل `fprintf` للكتابة في الملفّات.

القراءة من ملف

لدينا أيضاً ثلاث دوال للقراءة من ملف، اسمها مختلف قليلاً فقط عن دوال الكتابة :

- `fgetc` : قراءة محرف.
- `fgets` : قراءة سلسلة محرفيّة.
- `fscanf` : قراءة سلسلة منسّقة.

سأسرع قليلاً في شرح هذه الدوال : إذا كنت قد فهمت ما كتبته من قبل، فلن تجد أي صعوبة مع هذه الدوال.

fgetc

أولاً، النموذج :

```c
1 int fgetc(FILE* pointerOnFile);
```

هذه الدالة تقوم بإرجاع `int` : إنه الحرف الذي تمّت قراءته. إذا لم تقرأ أيّ محرف، فستعيد القيمة `EOF`.

> ؟ لكن كيف لنا أن نعرف الحرف الذي نقرؤه ؟ ماذا لو أردنا قراءة الحرف الثالث و أيضا العاشر، كيف نفعل هذا ؟

في الواقع، في كلّ مرة تقرأ فيها ملفّا، فهناك "مؤشر" (Cursor) (مثل المؤشّر الذي يغمز في محرر النصوص) يتحرّك في كلّ مرة. و هذا المؤشّر افتراضي طبعاً، لن تتمكن من رؤيته على الشاشة. و هو يشير إلى أين وصلنا في قراءة الملف.

سنتعلم لاحقاً كيف نعرف الوضعية التي وصل إليها المؤشّر بالضبط و أيضا كيف نحرّكه من مكانه (و ذلك لكي نقوم بتحريكه إلى بداية الملف مثلا، أو إلى مكان محرف محدّد، كالمحرف العاشر).

`fgetc` تقوم بتحريك المؤشر بحرف واحد في كلّ مرة تقرأ فيها واحدا. أي أنك إن استدعيت `fgetc` مرة ثانية، فستقرأ المحرف الثاني، ثم الثالث و هكذا. و بهذا يمكنك استعمال حلقة تكرارية لقراءة محارف الملف واحدا واحدا.

سنقوم بكتابة شفرة تقرأ كلّ محارف الملف واحدا واحدا و في كلّ مرّة تكتبها على الشاشة. الحلقة ستتوقف حينما تعيد `fgetc` القيمة `EOF` (و التّي تعني "End Of File" أي "نهاية الملف").

```
1  int main(int argc, char *argv[])
2  {
3          FILE* file = NULL;
4          int currentCharacter = 0;
5          file = fopen("test.txt", "r");
6          if (file != NULL)
7          {
8                  // A loop to read the characters one by one
9                  do
10                 {
11                         currentCharacter = fgetc(file); // Read the character
12                         printf("%c", currentCharacter); // Display it
13                 } while (currentCharacter != EOF); // Continue while fgets didn
                       't return EOF (End Of File)
14                 fclose(file);
15         }
16         return 0;
17 }
```

الكونسول ستقوم بإظهار محتوى الملف كاملاً، مثلا :

```
Hello, I'm the content of the file test.txt !
```

fgets

هذه الدالة تقوم بقراءة سلسلة من ملف. هذا يجنبك قراءة كلّ محارف الملف واحدا واحدا. الدالة تقرأ على الأكثر سطراً واحداً (تتوقف عند ملاقاة أول `\n`)، إن أردت قراءة العديد من الأسطر، فعليك استعمال حلقة.

هذا نموذج الدالة :

```
1  char* fgets(char* string, int nbOfCharsToRead, FILE* pointerOnFile);
```

هذه الدالة تتطلب معاملا خاصّا نوعاً ما، و لكنّه سيكون عمليّا جدّا : عدد المحارف التي نريد قراءتها. هذا ما يطلب من الدالة `fgets` التوقف عن قراءة السطر إذا كان يحوي أكثر من X من المحارف. الفائدة : هذا يسمح لنا بضمان عدم حدوث تجاوز في الذاكرة ! في الواقع، إذا كان حجم السطر أكبر من أن تسعه السلسلة المحرفيّة، فمن الممكن أن تقرأ عددا من المحارف أكثر ممّا يسمح به المكان المتوفّر، و هذا قد يسبب تعطّل البرنامج.

سنتعلّم كيف نقرأ سطراً واحداً باستخدام `fgets` ، (ثم بعدها سنرى كيفية قراءة ملف كامل).

لهذا فسنقوم بتعريف سلسلة محرفيّة كبيرة كفاية لتخزين السطر المراد قراءته (على الأقل نتمنّى ذلك، لا يمكننا أن نكون متأكّدين 100%). سترى فائدة استخدام الـ`#define` في تعريف حجم جدول :

```c
#define MAX_SIZE 1000 // A table of size 1000
int main(int argc, char *argv[])
{
    FILE* file = NULL;
    char string[MAX_SIZE] = ""; // Empty string of size MAX_SIZE
    file = fopen("test.txt", "r");
    if (file != NULL)
    {
        fgets(string, MAX_SIZE, file); // Read at maximum MAX_SIZE characters
           from the file, store them in  "string"
        printf("%s", string); // Display the string
        fclose(file);
    }
    return 0;
}
```

النتيجة هي نفسها النتيجة السابقة، مع العلم أنّ المحتوى يُكتب في الكونسول :

```
Hello, I'm the content of the file test.txt !
```

الفرق هو أننا هنا لم نستعمل حلقة تكرارية. نقوم بإسترجاع محتوى الملف كاملا في مرّة.

أنت تلاحظ بكلّ تأكيد الآن فائدة استعمال `#define` في شفرتك لتعريف الحجم الأقصى لجدول مثلاً. في الواقع، `MAX_SIZE` مستعمل في مكانين مختلفين في الشفرة :

- المرة الأولى لتعريف حجم الجدول الذي نريد إنشاءه.
- مرّة اخرى في الـ`fgets` لنقوم بتحديد عدد المحارف التي نقرؤها.

الفائدة هنا، هي أنّه في حال ما وجدت أن السلسلة المحرفيّة غير كبيرة كفاية لقراءة الملف، فلن يكون عليك سوى تعديل سطر الـ`#define` و إعادة الترجمة. هذا سيجنّبك البحث عن كلّ مكان من الشفرة وضعت فيه حجم الجدول. المعالج القبلي سيقوم باستبدال كل تكرار لـ`MAX_SIZE` بالقيمة الجديدة.

كما قلت فإن `fgets` تقرأ على الأكثر سطراً واحدا في المرّة. تتوقف عن قراءة السطر عندما تجاوز عدد المحارف الذي سمحت لها بقراءتها.

نعم و لكن : حاليّا، نحن لا نجيد سوى قراءة سطر واحد باستخدام `fgets` . كيف لنا أن نقرأ كل الملف ؟ الجواب بسيط : بحلقة تكرارية !

الدالة `fgets` تعيد `NULL` في حالة لم تستطع قراءة ما طلبته منها.
أي أن الحلقة يجب أن تنتهي بمجرّد أن تعيد `fgets` القيمة `NULL` .

ليس علينا سوى استعمال الحلقة `while` لكي نقوم بالتكرار مادامت `fgets` لم ترجع `NULL` :

```
#define MAX_SIZE 1000
int main(int argc, char *argv[])
{
    FILE* file = NULL;
    char string[MAX_SIZE] = "";
    file = fopen("test.txt", "r");
    if (file != NULL)
    {
        while (fgets(string, MAX_SIZE, file) != NULL) // Read the file while
            there's no error (NULL)
        {
            printf("%s", string); // Display the string that we've read
        }
        fclose(file);
    }
    return 0;
}
```

هذه الشفرة تقوم بقراءة الملف سطراً سطراً و إظهار الأسطر.

السطر الأكثر لفتاً للانتباه في الشفرة هو :

```
while (fgets(string, MAX_SIZE, file) != NULL)
```

سطر الـ `while` يقوم بأمرين : قراءة سطر من الملف و التأكد أن `fgets` لم تُعِد `NULL`. يمكن ترجمة هذا كالتالي : "اقرأ سطراً جديداً ما دمنا لم نصل إلى نهاية الملف".

fscanf

مبدأ هذه الدالة مشابه تماماً لمبدأ نظيرتها `scanf`، هنا أيضا.
هذه الدالّة تقوم بقراءة ملفّ تمت كتابته بشكل محدّد.

لنفترض أن الملف يحتوي على ثلاثة أعداد مفصولة بفراغ، و هي مثلا أكبر ثلاثة نقاط تم التحصل عليها في لعبتك : `15 20 30`.

أنت تريد أن تسترجع كلّ واحد من هذه الأعداد في متغير من نوع `int`.
الدالة `fscanf` ستسمح لك بالقيام بهذا بشكل سريع.

```
int main(int argc, char *argv[])
{
  FILE* file = NULL;
  int score[3] = {0}; // Table of the 3 best scores
  file = fopen("test.txt", "r");
  if (file != NULL)
  {
    fscanf(file, "%d %d %d", &score[0], &score[1],&score[2]);
```

```
9	    printf("The best scores are : %d, %d and %d", score[0], score[1], score[2])
	        ;
10	    fclose(file);
11	  }
12	  return 0;
13	}
```

```
The best scores are : 15, 20 and 30
```

كما ترى، فالدالة `fscanf` تنتظر ثلاث أعداد مفصولة بفراغ (`"%d %d %d"`). ستقوم بتخزينهم في جدولنا ذو الخانات الثلاث.

نقوم لاحقاً بإظهار كلّ القيم المسترجعة.

> حتّى الآن، لم استعمل سوى رمز `%d` واحداً في الدالة `scanf` .
> اليوم اكتشفتَ بأنه بإمكانك أن تستعمل العديد منها. إذا كان الملف مكتوبا بطريقة محدّدة جيدّا، فهذا يسمح لك بالإسراع لاسترجاع كلّ واحدة من هذه القيم.

3.16 التحرك داخل ملف

كنت قد كلّمتك عن وجود "مؤشّر" افتراضي (Virtual cursor) قبل قليل. سنقوم الآن بدراسته بشكل أكثر تفصيلاً.

في كلّ مرة تفتح فيها ملفا، فهناك مؤشّر يشير إلى وضعيتك في الملف. و لتتخيّله تماما مثل مؤشر محرر النصوص. يدلّ على المكان الذّي أنت فيه من الملف، أي أين ستقوم بالكتابة.

كتلخيص، نظام المؤشر يسمح لك بالكتابة و القراءة في وضعية محددة من الملف.

توجد ثلاث دوال لتتعرف عليها :

- `ftell` : تدلّنا على الوضعية التي نحن بها حالياً في الملف.
- `fseek` : تُموضع المؤشّر في مكان محدد.
- `rewind` : تقوم بإرجاع المؤشّر إلى بداية الملف (هذا مكافئ للطلب من الدالة `fseek` أن تموضع المؤشّر في البداية).

`ftell` : الموضع في الملف

هذه الدالة بسيطة الاستعمال جدّا. تعيد الموضع الذي يتواجد به المؤشّر حاليا بنوع `long` :

```
1	long ftell(FILE* pointerOnFile);
```

العدد الّذي يتم ارجاعه يدلّ على موضع المؤشّر في الملف.

التموضع داخل الملف : fseek

نموذج `fseek` هو التالي :

```c
int fseek(FILE* pointerOnFile, long deplacement, int origin);
```

الدالّة `fseek` تسمح بتحريك المؤشّر بعدد من المحارف (يدلّ عليها `deplacement`) انطلاقا من الموضع الذّي يدلّ عليه `origin`.

- العدد `deplacement` يمكن له أن يكون عدداً موجباً (للتقدم إلى الأمام)، معدوما (= 0) أو سالباً (للرجوع إلى الخلف).
- أمّا بالنسبة للعدد `origin` فهو يأخذ إحدى القيم التالية :
  - `SEEK_SET` تعني بداية الملف.
  - `SEEK_CUR` تعني الموضع الحالي نفسه.
  - `SEEK_END` تعني نهاية الملف.

إليك بعض الأمثلة لكي تفهم جيّدا كيف نتلاعب بـ `deplacement` و `origin` :

- هذه الشفرة تضع المؤشر محرفين بعد بداية الملف :

```c
fseek(file, 2, SEEK_SET);
```

- هذه الشفرة تضع المؤشّر أربع محارف قبل الوضعية الحالية :

```c
fseek(file, -4, SEEK_CUR);
```

لاحظ أن قيمة `deplacement` سالبة لأننا نحرّك إلى الوراء.

- الشفرة التالية تضع المؤشّر في نهاية الملف :

```c
fseek(file, 0, SEEK_END);
```

إذا كتبت، بعد القيام بـ `fseek` تحرّكك إلى نهاية الملف، فذلك سيضيف معلومات إلى نهاية الملف (الملف سيتمّ إكماله).
بالمقابل، إذا وضعت المؤشّر في بداية الملف وكتبت، فهذا سيستبدل النصّ الموجود هناك. لا توجد طريقة لـ"إدراج" نص في ملف. إلا إن قمت بنفسك ببرمجة دالة تقرأ المحارف لتتذكّرها قبل إستبدالها !

> ؟ لكن كيف لي أن أعرف أيّ موضع يجب أن أذهب إليه للقراءة و الكتابة في الملف ؟

هذا يعود إليك. إن كان ملفاً قمت أنت بكتابته، فأنت تعرف كيف تمّ بناءه. أنت تعرف أين تذهب للبحث عن المعلومة : مثلا، أحسن النتائج المسجلة في اللعبة في الموضع 0، أسماء آخر اللاعبين في الموضع 50، إلخ.

سنقوم بعمل تطبيقي لاحقاً حيث ستفهم، إذا لم تكن قد فهمت بالفعل الآن، كيف نذهب للبحث عن معلومة تهمّنا. لا تنس بأنّك أنت من يعرّف كيفيّة بناءه. إذن عليك أن تقول : "أضع نتيجة أحسن لاعب في السطر الأوّل، الخاصة بثاني أحسن لاعب في السطر الثاني، إلخ."

> ! الدالة `fseek` قد تتعامل بشكل غريب مع الملفات المفتوحة بوضع النص (Text mode). عادة، نحن نستعملها أكثر مع الملفات المفتوحة بالوضع الثنائي (Binary mode). عند القراءة و الكتابة في ملف بوضع النصّ، فإنّنا عادة ما نفعل ذلك حرفاً حرفاً. الشيء الوحيد الذي نسمح به غالبا في وضع النصّ مع `fseek` هو العودة إلى البداية أو التموضع في نهاية الملف فقط.

rewind : الرجوع إلى البداية

هذه الدالة مكافئة لاستخدام `fseek` لإرجاعنا إلى الموضع 0 في الملف :

```
1 void rewind(FILE* pointerOnFile);
```

طريقة الاستعمال بسيطة كالنموذج. أنت لست بحاجة إلى شرح إضافيّ.

4.16 إعادة تسمية و حذف ملف

ننهي هذا الفصل بنُعومة عن طريق دراسة دالتين بسيطتين للغاية :

- `rename` : إعادة تسمية ملف.
- `remove` : حذف ملف.

الشيء الخاصّ في هاتين الدالتين هو أنهما لا تحتاجان مؤشراً نحو الملف لكي تعملا. يكفيهما فقط اسم الملف المراد حذفه أو تغيير اسمه.

rename : إعادة تسمية ملف

إليك نموذج هذه الدالة :

```
1 int rename(const char* oldName, const char* newName);
```

الدالة تعيد 0 إذا نجحت في اعادة التسمية، و إلّا فستعيد قيمة مختلفة عن 0. هل من اللازم أن أعطيك مثالا ؟ إليك واحدا :

```
1 int main(int argc, char *argv[])
2 {
3     rename("test.txt", "test_rename.txt");
4     return 0;
5 }
```

remove : حذف ملف

هذه الدالة تقوم بحذف ملف دون ترك أي أثر :

```
int remove(const char* fileToDelete);
```

X

كن حذرا جداً عند استعمالك لهذه الدالة ! هي تحذف الملف بدون أن تطلب منك أيّ تأكيد ! الملف لن يوضع في سلة المحذوفات، بل يسحذف حرفيّا من القرص الصلب. لن يمكنك استعادة ملفّ محذوف بهذه الطريقة (إلّا باستعمال أدوات خاصّة باسترجاع الملفّات، لكنّ هذه العملية قد تكون طويلة، معقّدة و قد لا تنجح).

هذه الدالة مناسبة لإنهاء الفصل، فلم أعد في حاجة إلى الملف `test.txt`، يمكنني الآن حذفه :

```
int main(int argc, char *argv[])
{
    remove("test.txt");
    return 0;
}
```

الفصل 17

الجز الحيّ للذاكرة (Dynamic memory allocation)

كل المتغيّرات التي أنشأناها لحد الآن تمّ إنشاؤها تلقائيّا من طرف المترجم الخاصّ بلغة C. لقد كانت الطريقة البسيطة. رغم ذلك، توجد طريقة يدوية أكثر لإنشاء متغيّرات و نسمّيها بالحجز الحيّ (Dynamic allocation).

من بين فوائد الحجز الحيّ هو السماح لبرنامج بحجز مكان لازم لتخزين جدول في الذاكرة لا يُعرف حجمه قبل بداية الترجمة. في الواقع، حتّى الآن، كان حجم جداولنا ثابتاً في الشفرة المصدريّة. بعد قراءة هذا الفصل، ستستطيع إنشاء جداول بطريقة أكثر مرونة !

من الضروري أن نتقن التعامل مع المؤشرات لتتمكّن من قراءة هذا الفصل ! إن كانت لديك بعض الشكوك حول المؤشرات، أنصحك بالذهاب لإعادة قراءة الفصل الموافق قبل البدأ.

عندما نقوم بالتصريح عن متغيّر، فإننا نقول أننا طلبنا **حجز مكان في الذاكرة** :

```
1 int myNumber = 0;
```

عندما يصل المترجم إلى سطر مشابه للسطر السابق، يقوم بالأمور التالية :

- يقوم البرنامج بطلب إذن من نظام التشغيل (Windows، GNU/Linux، Mac OS ...) ليحجز شيئا من الذاكرة.
- يستجيب نظام التشغيل بإعطاء البرنامج عنوان الخانة حيث يمكنه تخزين المتغيّر (يعطيه العنوان الّذي حجزه له).
- عندما تنتهي الدالّة، المتغيّر يتم حذفه من الذاكرة. برنامجك يقول لنظام التشغيل : "أنا لم أعد بحاجة إلى المكان في الذاكرة الّذي حجزته في ذلك العنوان، شكرا ! التاريخ لا يحدّد إن كان البرنامج قد قال فعلا "شكرا" لنظام التشغيل، لكنّ هذا في مصلحته لأنّ نظام التشغيل هو الّذي يتحكّم في الذاكرة !

لحد الآن كل الأمور كانت تلقائيّة. عندما نصرّح عن متغير فإن نظام التشغيل يتمّ استدعاءه تلقائياً من طرف البرنامج. ما رأيك إذا بفعل هذا بطريقة يدوية ؟ ليس لأننا نريد أن نستمتع بفعل شيء معقّد، بل لأننا أحيانا نضطرّ لفعل ذلك !

في هذا الفصل سنقوم بـ :

- دراسة كيف تعمل الذاكرة (نعم، مرّة أخرى !) لنعرف ما الحجم الذي يحجزه كل متغيّر حسب نوعه.
- ثمّ ندخل في موضوعنا الأساسي : سنرى كيف نطلب من نظام التشغيل يدويّاً أن يحجز لنا مكانا في الذاكرة. هذا ما سنسميه الحجز الحيّ للذاكرة.
- و أخيراً، سنكتشف الفائدة من القيام بالحجز الحيّ بتعلّم إنشاء جدول ذي حجم غير معروف إلّا عند اشتغال البرنامج.

1.17 حجم المتغيرات

بحسب نوع المتغير التي نريد إنشاءه (`int`، `char`، `float` ...) فنحن نحتاج إلى حجم معيّن من الذاكرة.

في الواقع، لتخزين عدد من -128 إلى 127 (`char`) لن نحتاج إلا إلى بايت واحد من الذاكرة. هذا حجم صغير للغاية.
بالمقابل، `int` يحجز عادة حوالي 4 بايتات من الذاكرة. بينما `double` يحجز 8 بايتات.

المشكل هو ... أن هذا ليس صحيحا دائما. هذا يعتمد على الأجهزة : فقد يكون `int` يحجز 8 بايتات. من يعلم ؟
هدفنا هنا أن نتعرّف كم يحجز كلّ نوع من حجم في الذاكرة على حاسوبك.

توجد وسيلة سهلة جدّا لمعرفة هذا : استعمال العامل `sizeof()`.
على عكس الظاهر، فهو ليس دالة، بل عبارة عن إحدى الوظائف الأساسية من لغة الـC، يجب عليك فقط أن تضع بين القوسين النوع الذي تريد تحليله.
لمعرفة حجم `int` ، يجب كتابة التالي :

```
1 sizeof(int)
```

عند الترجمة، سيتم استبدال هذه الشفرة بعدد : عدد البايتات الّتي يحجزها `int` في الذاكرة. بالنسبة لي، `sizeof(int)` تساوي 4، و هذا يعني أنّ `int` يأخذ 4 بايتات. بالنسبة لك، ستكون نفس القيمة على الأرجح، لكنّها ليست قاعدة.
جرّب لترى، بعرض القيمة عن طريق `printf` مثلا :

```
1 printf("char : %d bytes\n", sizeof(char));
2 printf("int : %d bytes\n", sizeof(int));
3 printf("long : %d bytes\n", sizeof(long));
4 printf("double : %d bytes\n", sizeof(double))
```

بالنسبة لي ، هذا يظهر على الشاشة :

```
char : 1 bytes
int : 4 bytes
long : 4 bytes
double : 8 bytes
```

لم أختبر كل الأنواع الّتي نعرفها، أتركك لتجرّب أحجام الأنواع الأخرى.

أنت تلاحظ أن `int` و `long` يحجزان نفس الحجم من الذاكرة. إنشاء `long` يعود تماما إلى إنشاء `int`، هذا يأخذ 4 بايتات من الذاكرة.

في الواقع، النوع `long` هو مكافئ لنوع نسميه `long int`، و الذي هو مكافئ لنوع ... `int` نفسه. باختصار، فإن هذه أسماء كثيرة مختلفة لأجل أشياء ليست بالكبيرة، في النهاية ! امتلاك أنواع مختلفة كثيرة كان أمرا مهمّا في الوقت الذي لم تكن الحواسيب تملك كثيرا من ذاكرة. كنا نبحث دائما لاستخدام الحدّ الأدنى من الذاكرة باستخدام النوع المناسب.
اليوم، هذا لم يعد مفيدا كثيرا لأنّ ذاكرة الحاسوب صارت كبيرة جدّا. بالمقابل، هذه الأنواع لا تزال مفيدة إذا كنت تنشئ برامج للأنظمة المضمّنة (Embedded systems) حيث الذاكرة المتوفّرة أقل. أظن مثلا في البرامج الموجّهة للهواتف المحمولة، الأليّات، إلخ.

هل بإمكاننا أن نُظهر حجم نوع مخصّص قمنا نحن بإنشائه (هيكل) ؟

نعم ! `sizeof()` تعمل مع الهياكل أيضا !

```c
typedef struct Coordinates Coordinates;
struct Coordinates
{
        int x;
        int y;
};
int main(int argc, char *argv[])
{
        printf("Coordinates  : %d bytes\n", sizeof(Coordinates));
        return 0;
}
```

```
Coordinates : 8 bytes
```

كلما احتوى الهيكل من مركّبات كلّما أخذ حجما أكثر من الذاكرة. الأمر منطقي تماما، أليس كذلك ؟

طريقة أخرى للنظر إلى الذاكرة

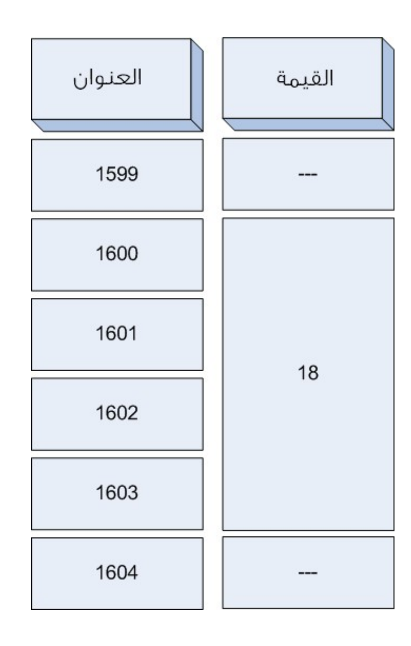
لحد الآن، كل المخططات التي قدّمتها لك عن الذاكرة لم تكن دقيقة. سنجعلها أخيرا دقيقة حقا و صحيحة بما أننا تعلّمنا الآن كم يأخذ كل نوع من حجم بالذاكرة.

إن صرّحنا عن متغير من نوع `int` :

```c
int number = 18;
```

و `sizeof(int)` يعطينا 4 بايت على حاسوبنا، هذا يعني أن المتغير يحجز 4 بايت في الذاكرة !

لنفترض أن المتغير `number` محجوز بالعنوان 1600 من الذاكرة. سيكون لدينا إذا المخطط التالي للذاكرة :



هنا، يمكننا فعلاً أن نرى بأن المتغير `number` من النوع `int` يحجز 4 بايت من الذاكرة. فهو يبدأ من العنوان 1600 و ينتهي عند العنوان 1603، المتغير القادم لن يتم تخزينه إلا إبتداءً من العنوان 1604 !

إن جربنا نفس الشيء مع `char` ، فالمتغير لن يأخذ سوى بايت واحد في الذاكرة (الشكل التالي) :

العنوان	القيمة
1599	---
1600	18
1601	---
1602	---
1603	---
1604	---

تخيّل الآن جدولا من `int` !

كل "خانة" من الجدول ستحجز 4 بايت. إن كان الجدول يحوي مثلاً 100 خانة :

```c
int table[100];
```

سنحجز إذن $400 = 4 * 100$ بايت في الذاكرة.

> ؟ ماذا لو كان الجدول فارغاً، هل سيحجز 400 بايت ؟

نعم بالطبع ! فالمكان في الذاكرة قد تمّ حجزه، و لا يملك أي برنامج الحقّ في استخدام هذه الخانات (غير هذا البرنامج). بمجرّد التصريح عن متغيّر، سيأخذ مكانه مباشرة المكان في الذاكرة.

لاحظ لو أننا ننشئ جدولا من نوع `Coordinates` :

```
Coordinates table[100];
```

سيستخدم هذه المرّة $800 = 8 * 100$ بايت.

من المهمّ الفهم الجيّد لهذه الحسابات البسيطة لنواصل بقيّة الفصل.

2.17 الحجز الحيّ للذاكرة

فلندخل إلى صلب الموضوع. سأذكّرك بهدفنا : تعلّم كيفيّة طلب الذاكرة يدوياً.

سنحتاج إلى تضمين المكتبة `stdlib.h`. إن كنت قد اتّبعت نصائحي، فقد ضمّنتها في كلّ براجك. هذه المكتبة تحتوي على دالّتين سنحتاج إليهما :

- `malloc` ("Memory ALLOcation" بمعنى "حجز الذاكرة") : تطلب الإذن من نظام التشغيل لاستخدام الذاكرة.
- `free` (تحرير) : تسمح للإشارة لنظام التشغيل بأننا لم نعد بحاجة إلى الذاكرة الّتي طلبناها. المكان في الذاكرة تمّ تحريره، يستطيع برنامج آخر الآن استخدامها عند الحاجة.

عندما تقوم بحجز يدوي للذاكرة، فعليك اتباع الخطوات التالية :

1. استدعاء `malloc` من أجل طلب الذاكرة.
2. اختبار القيمة التي تمّ إرجاعها من طرف `malloc` لمعرفة ما إن نجح نظام التشغيل في حجز الذاكرة.
3. ما إن ننتهي من استخدام الذاكرة، يجب علينا تحريرها باستعمال `free`. إن لم نفعل هذا، فسنتعرّض لتسريبات ذاكرة، أي أنّ البرنامج يخاطر بحجز كثير من الذاكرة مع أنّه ليس بحاجة إلى كلّ هذا المكان.

هل تتذكّرك هذه الخطوات الثلاث في فصل الملفات ؟ نعم يجب أن تفعل ! المبدأ واحد تماما : نحجز، نختبر إن نجح الحجز، ثمّ نحرر عندما ننتهي من الاستعمال.

malloc لنطلب الإذن لحجز الذاكرة

نموذج الدالة `malloc` هزليّ جدّاً، سترى :

```
void* malloc(size_t numberOfNecessaryBytes);
```

الدالة تأخذ معاملا واحدا : عدد البايتات الّتي يجب حجزها. هكذا، يكفي كتابة `sizeof(int)` لحجز مكان من أجل تخزين `int`.

و لكنّ الشيء الذي يثير الفضول، هو القيمة التي ترجعها الدالة : إنّها تعيد ... `void*` ! إذا لازلت تتذكّر فصل الدوال، كنت قد قلت لك بأن الكلمة `void` تعني "الفراغ" و نستعملها لنشير إلى أن الدالة لا تُعيد أية قيمة.

إذن هنا، لدينا دالة تُعيد ... "مؤشّراً نحو فراغ" ؟ هذه نكتة جيدة !
يبدو أن هؤلاء المبرمجين لديهم حسّ فكاهي متطوّر.

كن متأكّدا، يوجد سبب. في الحقيقة، هذه الدالة تعيد عنوان الخانة التي حجزها نظام التشغيل من أجل متغيّرك. إن استطاع النظام إيجاد مكان لك في العنوان 1600، فالدالة ستعيد مؤشّرا يحوي العنوان 1600.

المشكل هو أن الدالة `malloc` لا تعرف نوع المتغير التي نريد إنشاءه. في الواقع، أنت لا تعطيها سوى معامل واحد : عدد البايتات في الذاكرة الّتي تحتاجها. فإذا طلبت 4 بايت، فهذا يمكن أن يعني `int` أو ربما `long` مثلا !

بما أنّ `malloc` لا تعرف أيّ نوع يجب عليها أن تعيد، فهي تعيد النوع `void*`. سيكون مؤشّرا نحو أيّ نوع كان. يمكننا أن نقول أنّه مؤشّر جامع.

لننتقل إلى التطبيق.
إذا كنت أريد الاستمتاع بإنشاء متغير من نوع `int` يدويّا في الذاكرة، يجب أن أشير لـ`malloc` أنني أحتاج إلى `sizeof(int)` بايت في الذاكرة.
أسترجع قيمة `malloc` في مؤشر على `int` :

```
int* allocatedMemory = NULL; // Create a pointer on int
allocatedMemory = malloc(sizeof(int)); // The function malloc puts the
    allocated address in the pointer.
```

في نهاية هذه الشفرة، `allocatedMemory` هو مؤشّر يحتوي على عنوان حجزه نظام التشغيل لك، لنقل مثلا القيمة 1600 للإكمال من مخططاتي السابقة.

اختبار المؤشّر

الدالة `malloc` أعادت في المتغير `allocatedMemory` عنوان الخانة التي تم حجزها بالذاكرة. هناك احتمالان :

- إذا نجح الحجز، فالمؤشّر سيحتوي عنوانا.

• إذا فشل الحجز، فالمؤشّر سيحتوي العنوان `NULL`.

إنه من النادر أن تفشل عملية حجز الذاكرة، لكن هذا ممكن. تخيّل أنك تطلب حجز 34 Go من الذاكرة العشوائية، في هذه الحالة، ستفشل عملية الحجز على أغلب الظن.

من المستحسن دائماً أن نختبر ما إن تمت العملية بنجاح. سنفعل هذا : إن فشل الحجز، فهذا يعني أن المساحة الحرّة من الذاكرة العشوائية لم تكن كافية (هذه حالة حرجة). في حالة كهذه، يجب إيقاف البرنامج فورا لأنّه، على أية حال، لن يكون قادراً على الاستمرار بشكل عاديّ.

سنستعمل دالة قياسيّة لم يسبق لنا رؤيتها حتّى الآن : `exit()`. هذه الأخيرة توقف البرنامج فورا. إنّها تأخذ معاملا : القيمة الّتي يجب إعادتها من طرف البرنامج (هذا في الحقيقة يوافق الـ`return` الخاص بالـ`main`).

```
1  int main(int argc, char *argv[])
2  {
3          int* allocatedMemory = NULL;
4          allocatedMemory = malloc(sizeof(int));
5          if (allocatedMemory == NULL) // If the allocation has failed
6          {
7          exit(0); // Stop the program
8          }
9          // Else, we can continue the program normally.
10         return 0;
11 }
```

إذا كان المؤشر مختلفا عن `NULL`، يمكن للبرنامج أن يواصل العمل، و إلا فيجب إظهار رسالة خطأ أو حتّى إنهاء البرنامج لأنّه لن يتمكّن من الاستمرار بشكل صحيح إن لّم يكن هناك مكان في الذاكرة.

`free` : تحرير الذاكرة

مثلما استعملنا الدالة `fclose` لنغلق ملفاً لم نعد في حاجة إليه، سنستعمل الدالة `free` من أجل تحرير الذاكرة التي لم نعد بحاجة إليها.

```
1  void free(void* pointer);
```

الدالة `free` بحاجة فقط إلى عنوان الذاكرة المراد تحريرها. سنرسل لها إذن مؤشّرنا، أي `allocatedMemory` في مثالنا.
إليكم المخطط الكامل و النهائي، مشابه بشكل كبير لما رأينا في الفصل الخاص بالملفات :

```
1  int main(int argc, char *argv[])
2  {
3          int* allocatedMemory = NULL;
4          allocatedMemory = malloc(sizeof(int));
5          if (allocatedMemory == NULL) // Verify if the memory has been allocated
6          {
```

```
7            exit(0); // Error: Stop everything !
8        }
9        // We can use the memory here
10       free(allocatedMemory); // No need for the memory anymore, free it.
11       return 0;
12   }
```

مثال استخدام واقعي

سنبرمج شيئا درسناه منذ زمن طويل : الطلب من المستخدم تزويدنا بعُمره ثمّ عرضه له. الشيء المختلف عمّا كنّا نفعله سابقا هو أن المتغيّر هنا سيتمّ حجزه يدويا (نقول أيضا حيويّا) و ليس تلقائيّا كالسابق. إذن نعم، في المرّة الأولى، الشفرة أصعب قليلا. لكن قم بالمجهود اللازم لفهمها جيّدا، هذا ضروريّ :

```
1   int main(int argc, char *argv[])
2   {
3       int* allocatedMemory = NULL;
4       allocatedMemory = malloc(sizeof(int)); // Allocation of the memory
5       if (allocatedMemory == NULL)
6       {
7           exit(0);
8       }
9       // Using the memory
10      printf("How old are you ? ");
11      scanf("%d", allocatedMemory);
12      printf("You are %d years old\n", *allocatedMemory);
13      free(allocatedMemory); // Freeing the memory
14       return 0;
15  }
```

```
How old are you  ? 31
You are 31 years old
```

> حذار : بما أن `allocatedMemory` هو مؤشّر، فلا نستعمله بنفس الطريقة التي نستعمل بها متغيّرا حقيقيّا. للحصول على قيمة المتغير يجب وضع نجمة أمامه : `*allocatedMemory` (لاحظ `printf`). بينما للحصول على العنوان، يكفي فقط أن كتابة اسم المؤشر `allocatedMemory` (لاحظ `scanf`).
> كلّ هذا تم شرحه في فصل المؤشّرات. رغم ذلك، أعرف أن هذا سيأخذ وقتا و من الممكن أن تخلط بينهما. إن كانت هذه حالتك، فعليك بإعادة قراءة فصل المؤشّرات، فهو أساسي.

لنعد إلى الشفرة. لقد قمنا بحجز حيّ لمتغيّر من نوع `int`. في النهاية، ما كتبناه يعود تماما لاستخدام الطريقة "التلقائيّة" التي نعرفها الآن جيّدا :

```c
int main(int argc, char *argv[])
{
        int myVariable = 0; // Allocation of the memory (automatically)
        // Using the memory
        printf("How old are you ? ");
        scanf("%d", &myVariable);
        printf("You are %d years old\n", myVariable);
        return 0;
} // Freeing the memory (automatically at the end of the function)
```

```
How old are you ? 31
You are 31 years old
```

كملخص، لدينا طريقتان لإنشاء متغير، أي لحجز الذاكرة. إمّا أن نقوم بذلك :

- تلقائيّا : هي الطريقة الّتي تعرفها و التي استعملناها لغاية الآن.
- يدويا (حيويّا) : هي الطريقة الّتي أعلّمك إيّاها في هذا الفصل.

؟ أنا أجد أن الطريقة الحيّة معقّدة و بلا فائدة !

أكثر تعقيدا ... بالتأكيد. لكن بدون فائدة، لا ! أحيانا نكون مجبرين على حجز الذاكرة يدويّا كما سنرى الآن.

3.17 الحجز الحيّ لجدول

لحد الآن استعملنا الحجز الحي لإنشاء متغيّر صغير. عادة لا نستخدم الحجز الحيّ في هذا. نستخدم الطريقة التلقائيّة الّتي هي أبسط.

متى نحتاج للحجز الحيّ، تتساءلون ؟ أكثر شيء، نستخدمه لإنشاء جدول لا نعرف حجمه قبل تشغيل البرنامج.

لنتخيل مثلا برنامجاً يقوم بتخزين أعمار أصدقاء المستخدم في جدول، يمكنك فعل ذلك هكذا :

```c
int friendsAge[15];
```

لكن من قال أنه لديه 15 صديقا ؟ ربما لديه أكثر من هذا ! عندما تكتب الشفرة المصدريّة، لا يمكنك معرفة حجم الجدول قبل التشغيل، عندما تطلب من المستعمل إدخال عدد الأصدقاء.

فائدة الحجز الحيّ موجودة هنا : تطلب من المستخدم إدخال عدد الأصدقاء، ثم تنشئ جدولا لديه تماما الحجم اللازم (لا أصغر و لا أكبر). إن كان للمستخدم 15 صديقا، فسننشئ جدولا من 15 `int`، إن كان لديه 28، فسننشئ جدولا من 28 `int`، إلخ.

كما علّمتك، من الممنوع في لغة الـC إنشاء جدول بتحديد حجمه باستخدام متغيّر :

```c
int friends[friendsNumber];
```

> هذه الشفرة قد تعمل في بعض المترجمات لكن في حالات معيّنة، من المنصوح به عدم استخدامها !

فائدة الحجز الحيّ، هي أنّه يمكننا من إنشاء جدول حجمه تماما المتغيّر `friendsNumber`. و هذا بفضل شفرة تعمل في كلّ مكان !

سنطلب من `malloc` حجز `friendsNumber * sizeof(int)` بايت في الذاكرة :

```c
friends = malloc(friendsNumber ▫ sizeof(int));
```

هذه الشفرة تسمح بإنشاء جدول من نوع `int` حجمه يوافق تماما عدد الأصدقاء !

هذا ما يقوم به البرنامج بالترتيب :

1. نطلب من المستخدم كم من صديق لديه.
2. إنشاء جدول من `int` ذو حجم يساوي عدد أصدقائه (باستخدام `malloc`).
3. نطلب كم عمر كلّ واحد من أصدقائه واحدا واحدا، و نقوم بتخزينها في الجدول.
4. نظهر أعمار الأصدقاء من محتوى الجدول لنتأكد بأننا خزّنّا كلّ شيء.
5. في النهاية، و بما أننا لسنا بحاجة إلى الجدول الذي يحوي أعمار الأصدقاء، نقوم بتحريره بالدالة `free`.

```c
int main(int argc, char ▫argv[])
{
  int friendsNumber = 0, i = 0;
  int▫ friendsAge = NULL; // We use it after calling malloc
  // Request for friends count
  printf("How many friends do you have ? ");
  scanf("%d", &friendsNumber);
  if (friendsNumber > 0) // The user must have at least one friend (or i will
     be upset :p)
  {
    friendsAge = malloc(friendsNumber ▫ sizeof(int)); // Allocate the memory
       for the table
    if (friendsAge == NULL) // Verify the allocation
    {
      exit(0); // Stop everything
    }
    // Request for the ages one by one
    for (i = 0 ; i < friendsNumber ; i++)
    {
      printf("How old is the friend number %d ? ", i + 1);
      scanf("%d", &friendsAge[i]);
```

```
20      }
21      // Display the stored ages
22      printf("\n\nYour friends have the next ages :\n");
23      for (i = 0 ; i < friendsNumber ; i++)
24      {
25        printf("%d years\n", friendsAge[i]);
26      }
27      // Free the memory
28      free(friendsAge);
29    }
30    return 0;
31  }
```

```
How many friends do you have ? 5
How old is the friend number 1 ? 16
How old is the friend number 2 ? 18
How old is the friend number 3 ? 20
How old is the friend number 4 ? 26
How old is the friend number 5 ? 27
Your friends have the next ages :
16 years
18 years
20 years
26 years
27 years
```

هذا البرنامج عديم الفائدة : يطلب الأعمار ثمّ يعرضها بعد ذلك. لقد اخترت فعل هذا لأنّه مثال "بسيط" (هذا إن فهمت `malloc`).

أؤكّد لك : في بقيّة هذا الكتاب ستكون لنا فرص لاستخدام `malloc` في أمور أكثر إفادة من تخزين أعمار الأصدقاء !

ملخّص

- كل متغير يحجز مكانا مختلفا في الذاكرة و هذا حسب نوعه.
- يمكننا معرفة عدد البايتات التي يشغلها كل نوع باستعمال العامل `sizeof()`.
- الحجز الحيّ هو عبارة عن حجز يدوي لمكان في الذاكرة من أجل متغير أو من جدول.
- الحجز الحيّ يتم باستعمال الدالة `malloc()` و يجب خاصّة عدم نسيان تحرير الذاكرة باستعمال `free()` بمجرّد الانتهاء من الاستخدام.
- الحجز الحيّ يسمح بشكل أساسيّ بتعريف جدول حجمه يتمّ تعيينه بمتغيّر في حين تشغيل البرنامج.

الفصل 18

برمجة لعبة الـPendu

أكرر دائما : التطبيق شيء ضروريّ. هو ضروريّ لك لأنك اكتشفت كثيرا من المفاهيم النظرية و، أيّا كان ما تقول، لن تفهمها حقّا بدون تطبيق.

في هذا العمل التطبيقي، أقترح عليك إنشاء لعبة الـPendu. و هي لعبة حروف تقليديّة يتمّ فيها تخمين كلمة سريّة حرفا بحرف. الـPendu سيكون إذن لعبة في الكونسول بلغة C.

الهدف هو جعلك تستخدم كلّ ما تعلّمته حتّى الآن : المؤشرات، السلاسل المحرفيّة، الملفات، الجداول ... باختصار، الأشياء الجيّدة فقط !

1.18 التعليمات

سأقوم بشرح قواعد الـPendu الواجب إنشاءه. سأعطيك هنا التعليمات، أي سأشرح لك بدقّة كيف يجب أن تعمل اللعبة التي ستُنشئها.

أعتقد أن الجميع يعرف الـPendu، أليس كذلك ؟ هيّا، تذكير صغير لا يمكن أن يحدث ضررا : هدف الـPendu هو إيجاد الكلمة المخبّأة في أقلّ من عشر محاولات (يمكنك تغيير العدد الأقصى لتغيير صعوبة اللعبة، بالطبع !).

سريان الجولة

فلنفترض أن الكلمة المخبّأة هي RED.
ستقوم باقتراح حرف على الحاسوب، مثلا الحرف A. سيتأكّد الحاسوب ما إن كان هذا الحرف موجوداً في الكلمة المخفيّة.

> تذكّر : هناك دالة جاهزة في `string.h` تقوم بالبحث عن حرف في كلمة ! و بالطبع أنت لست مجبراً على استخدامها (شخصيّا، أنا لم أفعل).

انطلاقاً من هنا، يوجد احتمالان :

- الحرف موجود بالفعل في الكلمة : سنكشف مكان الحرف في الكلمة.

• الحرف غير موجود في الكلمة (هذا هو الحال هنا، لأن A ليس موجوداً في الكلمة RED) : سنخبر اللاعب بأن الحرف هذا غير موجود في الكلمة، و سننقص عدد المحاولات المتبقّية. عندما لا تبق أية محاولة (0 محاولة)، ستنتهي اللعبة و سيخسر.

> م
> في لعبة Pendu "حقيقة"، يفترض وجود شخص يتأسّف في كلّ مرّه نخطئ فيها. في الكونسول، سيكون من الصعب كثيرا رسم شخص يتأسّف بواسطة لاشيء غير النص، لذا سنكتفي بعرض جملة بسيطة مثل "بقي لك X محاولات قبل الموت الأكيد".

فلنفرض الآن أن اللاعب أدخل الحرف D. هذا الحرف موجود في الكلمة المخفيّة، لهذا لن نقوم بإنقاص عدد المحاولات المتبقّية للاعب. سنقوم بإظهار الكلمة مع الحروف الّتي تم إيجادها، أي شيء كهذا :

```
Secret word : □□D
```

إذا أدخل اللاعب فيما بعد الحرف R، و بما أنّه موجود في الكلمة، سنضيف الحرف إلى قائمة الحروف التي تم إيجادها و يتم إظهار الكلمة مع الحروف الّتي تمّ اكتشافها :

```
Secret word : R□D
```

حالة وجود حرف مكرر

في بعض الكلمات، يمكن أن نجد حرفاً مكرراً مرتين أو ثلاث، أو ربّما أكثر !
مثلا : يوجد إثنان من Z في كلمة PUZZLE، و كذلك يوجد ثلاثة E في كلمة ELEMENT.

ماذا علينا أن نفعل في حالة كهذه ؟ قواعد Pendu واضحة : إذا أدخل اللاعب الحرف E، كلّ حروف E في كلمة ELEMENT يجب أن تظهر دفعة واحدة :

```
Secret word : E□E□E□□
```

يعني أنه ليس على اللاعب أن يدخل 3 مرات الحرف E ليتم إكتشاف كل تكرار له في الكلمة.

مثال عن جولة كاملة

هذا ما ستبدو عليه جولة كاملة في الكونسول عند انتهاء البرنامج :

```
Welcome !
You have 10 remaining tries
What's the secret word ? □□□□
Suggest a letter : B
You have 9 remaining tries
What's the secret word ? □□□□
Suggest a letter : F
```

```
You have 9 remaining tries
What's the secret word ? F***
Suggest a letter : D
You have 9 remaining tries
What's the secret word ? F**D
Suggest a letter : O
You win ! The secret word is  : FOOD
```

قراءة حرف من الكونسول

قراءة حرف من الكونسول هي أكثر تعقيداً ممّا تبدو.
بديهيّا، لاسترجاع محرف، يفترض أنّك تفكّر في :

```c
scanf("%c", &myLetter);
```

و تماما، هذا جيّد. `%c` تعني أننا ننتظر محرفاً، و الذي سنقوم بتخزينه في `myLetter` (متغيّر من نوع `char`).

كل شيء يعمل جيداً ... ما دمنا لم نقم بـ`scanf` مرّة اخرى. يمكنك تجريب الشفرة التالية :

```c
int main(int argc, char* argv[])
{
        char myLetter = 0;
        scanf("%c", &myLetter);
        printf("%c", myLetter);
        scanf("%c", &myLetter);
        printf("%c", myLetter);
        return 0;
}
```

يفترض بهذه الشفرة أن تطلب حرفاً و تظهره، و ذلك لمرّتين.
جرّب. ما الذي يحصل ؟ تدخل حرفا، نعم، و لكن ... البرنامج يتوقّف مباشرة بعدها، فهو لا يطلب منك المحرف الثاني ! و كأنه تم تجاهل `scanf` الثانية.

ما الذي حصل ؟

في الواقع، حينما تدخل نصاً في الكونسول، فإن كل ما قمت بإدخاله يتمّ تخزينه في الذاكرة، بما في ذلك الزر Enter (`\n`).

لذلك، في أوّل مرّة تدخل فيها حرفا (A مثلاً) ثمّ تضغط على *Enter* فإن الحرف A هو من يتم إعادته من طرف `scanf` . بينما في المرّة الثانية، `scanf` سيعيد `\n` الموافق لـ*Enter* الذّي أدخلته سابقا !

لتجنب هذا، من الأحسن أن نكتب بأنفسنا دالتنا الخاصّة الصغيرة `readCharacter()` :

```c
char readCharacter()
{
  char character = 0;
  character = getchar(); // Read the first character
  character = toupper(character); // Convert the character to uppercase
  // Read other characters until reaching \n (to erase them)
  while (getchar() != '\n') ;
  return character; // Return the first character that have been read
}
```

هذه الدالة تستخدم `getchar()` الّتي هي دالة من `stdio.h` و هذا يعود تماماً إلى كتابة `scanf("%c", &letter);`. الدالة `getchar()` تقوم بإرجاع الحرف الذي قام اللاعب بإدخاله.

بعد ذلك، أستعمل أيضاً الدالة القياسيّة التي لم تسنح لنا فرصة تعلّمها في كتابنا : `toupper()`. هذه الدالّة تحوّل الحرف المعطى إلى كبير (Uppercase). هكذا، اللعبة ستعمل حتى إن أدخل اللاعب حروفاً صغيرة. يجب تضمين `ctype.h` لتستطيع استخدام هذه الدالة (لا تنس ذلك !).

تأتي بعد ذلك المرحلة الأكثر أهمية : و هي أن نقوم بمسح المحارف التي يمكن أن نكون قد أدخلناها. في الواقع، بإعادة استدعاء `getchar` نحصل على الحرف الثاني الذي تمّ إدخاله (مثلا `\n`).
ما أقوم به بسيط و يأخذ سطرا واحدا : أستدعي الدالة `getchar` في حلقة تكرارية حتى الوصول إلى `\n`. تتوقف الحلقة إذن، و هذا يعني أننا "قرأنا" كلّ المحارف الأخرى، سيتمّ إذن إفراغها من الذاكرة. نقول أنّنا **نفرغ المتغير المؤقت (Buffer)**.

> لماذا توجد فاصلة منقوطة في نهاية الـ `while` و لماذا لا نرى أية حاضنة ؟

في الواقع، استعملت حلقة تكرارية لا تحتوي على تعليمات (التعليمة الوحيدة، هي `getchar` داخل القوسين).
الحاضنتان ليستا ضروريّتين نظرا لأنه ليس لدينا ما نفعله غير `getchar`. لهذا أضع فاصلة منقوطة لتعويض الحاضنتين.
هذه الفاصلة المنقوطة تعني "لا تفعل شيئاً في كلّ دورة للحلقة". هذا أمر غريب قليلا، لكنها تقنيّة يجب معرفتها، تقنيّة يستعملها المبرمجون لإنشاء حلقات بسيطة و قصيرة.

اعلم أنّ الـ `while` كان بالإمكان كتابتها هكذا :

```c
while (getchar() != '\n')
{

}
```

لا يوجد شيء داخل الحاضنتين، إنّها اختياريّة، نظرا لأنّه ليس هناك شيء آخر لفعله. تقنيّتي الّتي تقتضي وضع فاصلة منقوطة فقط أبسط من تلك الخاصّة بالحاضنتين.

أخيرا، تقوم الدالّة `readCharacter` بإرجاع الحرف الأوّل الذي قمنا بقراءته : المتغيّر `character`.

خلاصة القول، في شفرتك، لا تستعمل :

```
scanf("%c", &myLetter);
```

و إنما استعمل بدل ذلك دالّتنا الرائعة :

```
myLetter = readCharacter();
```

قاموس الكلمات

لتجربة أولية للشفرة الخاصة بك، أطلب منك أن تقوم بتثبيت الكلمة السريّة مباشرة في الشفرة. أكتب مثلا :

```
char secretWord[] = "RED";
```

طبعا ستبقى الكلمة السريّة نفسها دائما إن تركناها هكذا، هذا ليس ممتعا. لكنّي طلبت منك فعل ذلك لكي لا تخلط المشاكل. في الواقع، عندما تعمل لعبة Pendu جيّدا (و فقط ابتداء من هذه اللحظة)، يمكنك البدء بالطور الثاني : إنشاء قاموس الكلمات.

> ما هو هذا "قاموس الكلمات" ؟

هو ملف يحتوي كثيرا من الكلمات للعبتك Pendu. يجب أن تكون كل كلمة على سطر. مثلا :

```
HOUSE
BLUE
AIRPLANE
XYLOPHONE
BEE
BUILDING
WEIGHT
SNOW
ZERO
```

في كل جولة جديدة، يجب على برنامجك أن يفتح الملف، و يأخذ كلمة عشوائية من القائمة. بفضل هذه الطريقة، سيكون لديك ملف يمكنك التعديل عليه كلّما أردت من أجل إضافة كلمات سريّة ممكنة من أجل Pendu.

> ستلاحظ أنني منذ البداية تعمّدت كتابة كلّ الكلمات بالحروف الكبيرة. في الواقع، في Pendu لا يتم التمييز بين الحروف الكبيرة و الحروف الصغيرة، و لهذا فمن المستحسن أن نقول منذ البداية : "كل حروف كلمات اللعبة كبيرة". عليك أن تنبّه اللاعب، في دليل استخدام اللعبة مثلا، أنه يفترض به إدخال حروف كبيرة لا صغيرة. بالمقابل، تعمّد تجنب العلامات الصوتية (accents) لتبسيط اللعبة (إن بدأنا اختبار é، è، ê، ë ... فلن ننتهي أبداً !). عليك إذن أن تكتب كلماتك كلّها بحروف كبيرة و بدون علامات صوتيّة.

المشكل الذي سيحدث لك سريعا هو أنه عليك معرفة عدد الكلمات الموجودة في القاموس. في الواقع، إن أردت إختيار كلمة عشوائية، يجب أن يتم أخذ عدد بين 0 و X، و أنت لا تعرف في بادئ الأمر كم من الكلمات يحتوي الملف.

لحلّ هذا المشكل، يوجد حلّان. يمكنك أن تشير في السطر الأول من الملفّ إلى عدد الكلمات الّتي يحويها :

```
3
HOUSE
BLUE
AIRPLANE
```

إلا أن هذه الطريقة مملة، لأنه يجب إعادة حساب عدد الكلمات يدويا في كلّ مرّة تضيف فيها كلمة (أو إضافة 1 إلى هذا العدد إن كنت ماكرا بدل إعادة الحساب، لكنّها تبقى طريقة بدائيّة قليلا). لهذا، أقترح عليك أن تعدّ تلقائيّا عدد الكلمات عن طريق قراءة الملف مرّة أولى باستخدام برنامجك. معرفة كم يوجد من كلمات أمر بسيط : عليك عدّ الـ `\n` (العودة إلى السطر) في الملف.

حينما تقرأ الملفّ في مرّة أولى لعدّ `\n`، فعليك القيام بـ `rewind` للعودة إلى البداية. لن يكون عليك إذن سوى أخذ عدد عشوائيّ بين عدد الكلمات الّتي عددتها، ثمّ عليك تخزين هذه الكلمة في سلسلة محرفيّة في الذاكرة.

سأتركك قليلا لتفكّر في كلّ هذا، لن أساعدك أكثر، و إلّا فلن يكون عملا تطبيقيا ! و اعلم بأن كلّ المعارف الّتي تحتاجها موجودة في الفصول السابقة، فأنت قادر تماما على إنشاء هذه اللعبة. إنه يتطلّب منك بعض الوقت و هو أقلّ سهولة ممّا يبدو عليه، و لكن إذا نظّمت الأمور جيّدا (بإنشاء قدر كاف من الدوال) سوف تصل.

بالتوفيق !

2.18 التصحيح (1 : شفرة اللعبة)

بقراءتك لهذه السطور، يعني أنك قد أكملت البرنامج، أو أنك لم تستطع إكماله.

لقد استغرقت شخصيّا وقتا أكبر ممّا كنت أعتقد في إنشاء هذه اللعبة البسيطة للغاية. هكذا دائما : نقول "هذا بسيط"، لكن في الحقيقة توجد الكثير من الحالات لدراستها.

رغم ذلك أصرّ على القول بأنك قادر على فعل هذا. يلزمك فقط بعض الوقت (بضع دقائق، بضع ساعات بضع أيام ؟)، لكنّنا لم نكن أبدا في سباق. أنا أفضّل أن تأخذ كثيرا من الوقت للوصول إلى الحل على ألّا تجرّب سوى 5 دقائق و ترى التصحيح.

لا تعتقد أنّي كتبت البرنامج من المحاولة الأولى. أنا أيضا، كنت أعمل خطوة بخطوة. بدأت بشيء بسيط جدّا، ثمّ شيئا فشيئا حسّنت الشفرة للوصول إلى النتيجة النهائيّة.
قمت بعدّة أخطاء أثناء كتابة الشفرة : نسيت في لحظة ما تهيئة متغير بشكل صحيح، نسيت كتابة نموذج دالة و كذلك حذف متغير لم يعد مفيدا في شفرتي. و حتى أنّي -أعترف- نسيت فاصلة منقوطة سخيفة في لحظة ما عند نهاية تعليمة.

لماذا أقول كل هذا ؟ لكي أخبرك أنّني لست معصوما من الأخطاء و أنّي أواجه تقريبا نفس المشاكل مثلك ("أيّها البرنامج البائس، هل ستعمل أم لا ؟!").

سأعرض عليك الحلّ على جزئين.

- أوّلا سأريك كيف أنشأت شفرة اللعبة نفسها، بتثبيت الكلمة المخفيّة مباشرة في الشفرة. اخترت الكلمة YELLOW لأنّها تسمح باختبار ما إن كنت تعاملت جيّدا مع المحارف المتكرّرة.
- بعد ذلك، سأريك كيف أضفت العمل بقاموس الكلمات لإعطاء كلمة سرّية عشوائيّة للاعب.

بالطبع، يمكنني أن أريك الشفرة دفعة واحدة و لكن ... سيكون هذا كثيرا في مرّة واحدة، و البعض لن تكون لديه الشجاعة لمحاولة فهم الشفرة.

سأحاول أن أشرح لك خطوة بخطوة طريقة عملي. تذكّر أنّ ما يهم، ليس النتيجة، و إنّما طريقة التفكير.

تحليل الدالة main

مثلما يعلم الجميع، كلّ شيء يبدأ بـ `main`. يجب ألا ننسى تضمين المكتبات `stdio`، `stdlib` و `ctype` (من أجل الدالة `toupper`) التي سنحتاج إليها أيضا :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <ctype.h>
4
5 int main(int argc, char* argv[])
6 {
7   return 0;
8 }
```

حسنا، لحدّ الآن يجب على الجميع أن يتابعوا.
الدالة `main` ستشكّل معظم اللعبة و ستقوم باستدعاء بعض الدوال حينما تحتاج إليها.

فلنبدأ بتعريف المتغيرات الضروريّة. كن متأكّدا، لم أفكّر في كلّ هذه المتغيرات من الوهلة الأولى، و لقد كان هناك أقلّ من هذا العدد في أوّل مرّة كتبت فيها الشفرة !

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <ctype.h>
4  int main(int argc, char* argv[])
5  {
6    char letter = 0; // Stores the letter suggested by the user
7    char secretWord[] = "YELLOW"; // The word that the user must find
8    int foundLetter[6] = {0}; // Boolean table. Each cell corresponds to a letter
         in the secret word. 0 = letter not found, 1 = letter found
9    int remainingTries = 10; // Counting the remaining tries (0 = dead)
10   int i = 0; // A little variable to browse the table
11   return 0;
12 }
```

لقد كتبت بمحض إرادتي تصريح كلّ متغير على سطر و وضعت كثيرا من التعليقات لشرح دور كل متغير. عمليّا، لست مضطرّاً إلى وضع كلّ هذه التعليقات كما يمكنك وضع الكثير من التصريحات في نفس السطر.

أعتقد أن أغلب المتغيرات تبدوا منطقية : المتغير `letter` يخزّن الحرف الذي يدخله المستخدم في كلّ مرة، `secretWord` يحوي الكلمة الواجب اكتشافها. `remainingTries` يحتوي عدد المحاولات المتبقّية، إلخ. المتغيّر `i` هو متغير صغير استعمله كي أتصفّح الجدول مستعملا الحلقة `for` . فهو ليس مهمّا جدّا لكنّه ضروري إذا أردنا القيام بحلقات.

و أخيراً المتغير الّذي يجب التفكير فيه، و الذي سيُمثّل الفرق، إنّه عبارة عن جدول من القيم المنطقية `foundLetter` . ستلاحظ بأنّي جعلت حجم الجدول يساوي عدد حروف الكلمة السريّة (6). هذا ليس أمراً عشوائياً : إذ أن كلّ خانة من جدول القيم المنطقية تمثّل حرفاً من الكلمة السرية. هكذا، الخانة الأولى تمثّل الحرف الأوّل، الثانية الحرف الثاني، إلخ. كلّ خانات الجدول مهيّئة في البداية على 0، و التي تعني "الحرف لم يتم إيجاده بعد". بتقدّم اللعبة، الجدول سيتمّ تعديله. من أجل كلّ حرف تمّ إيجاده من الكلمة، الخانة التي توافقها من `foundLetter` ستأخذ 1.

مثلا، إذا كان في مرحلة من الجولة، لدينا العرض "Y*LL*W"، فإن جدول الـ`int` سيحوي القيم : 101101 (1 لكل حرف تمّ إيجاده).
هذه الطريقة تسهّل علينا معرفة متى يربح اللاعب : يكفي التحقّق من أن جميع خانات الجدول لا تحوي سوى 1. في الحالة الأخرى، سيخسر اللاعب إذا وصل العدّاد `remainingTries` إلى 0.

فلننتقل إلى التالي :

```
printf("Welcome !\n\n");
```

هذه رسالة ترحيب، لا يوجد أي شيء مثير فيها. بالمقابل، الحلقة الرئيسيّة هي الأكثر أهميّة :

```
while (remainingTries  > 0 && !win(foundLetter))
{
```

اللعبة تستمرّ مادام قد بقي بعض المحاولات (`remainingTries > 0`) و اللاعب لم يربح.
إذا لم تبق له أية محاولة، فهذا يعني أنه فشل. إن ربح، فهذا يعني ... أنّه ربح. في كلتا الحالتين، يجب إيقاف اللعبة، أي إيقاف الحلقة التي تطلب قراءة حرف في كلّ مرة.

`win` هي دالة تقوم بتحليل الجدول `foundLetter` . تقوم بإعادة "صحيح" (1) إذا كان اللاعب قد ربح (أي أن الجدول `foundLetter` لا يحوي سوى 1)، "خطأ" (0) إن كان لم يربح بعد. لن أشرح لك الآن عمل الدالة بشكل مفصل، سنرى ذلك لاحقاً. حاليّا، يجب عليك فقط معرفة ما تفعله.

باقي الشفرة :

```
printf("\n\nYou have %d remaining tries", remainingTries);
printf("\nWhat's the secret word ? ");
for (i = 0 ; i < 6 ; i++)
{
  if (foundLetter[i]) // If the letter n° i has been found
```

```
6      printf("%c", secretWord[i]); // Display it
7    else
8      printf("□"); // Else, display □ for the letters that are not found
9  }
```

نقوم في كل مرة بإظهار عدد المحاولات المتبقّية و كذا الكلمة السريّة (مخفيّة بـ* بالنسبة للحروف التي لم يتمّ إيجادها).
يتم إظهار الكلمة السريّة المخفيّة بـ* بفضل حلقة `for` حيث أننا نحلّل كلّ حرف لنرى إن تمّ إيجاده (`if(foundLetter[i])`).
إن كان الشرط محققاً، سنظهر الحرف، و إلا سنظهر * لإخفاءه.

الآن بعدما أظهرنا ما يجب، سنطلب من اللاعب أن يدخل حرفاً جديداً :

```
1 printf("\nSuggest a letter : ");
2 letter = readCharacter();
```

أستدعي دالتنا `readCharacter()` . هذه الدالة تقرأ الحرف الأول الذي تمّ إدخاله، تجعله كبيراً ثم تفرّغ المتغير المؤقّت، أي أنّها تمسح بقيّة الحروف التي يمكن أن تبقى في الذاكرة.

```
1 // if it's NOT the right letter
2 if (!findLetter(letter, secretWord, foundLetter))
3   {
4     remainingTries--; // Decrement the remaining tries
5   }
6 }
```

نختبر ما إن كان الحرف الذي تمّ إدخاله موجودا في `secretWord` . نستدعي لأجل هذا دالّة أنشأناها تسمّى `findLetter` . سنرى بعد قليل شفرة هذه الدالّة.
حاليّا، كلّ ما يجب أن تعرفه، هو أنّ هذه الدالّة تعيد "صحيح" إن كان الحرف موجودا في الكلمة، "خطأ" إن لم تجده.

كما تلاحظ فالـ`if` يبدأ بعلامة تعجّب `!` و التي تعني "لا". الشرط يُقرأ إذن بهذه الطريقة : "إذا لم يتم إيجاد الحرف".
ماذا نفعل في حالة عدم إيجاد الحرف ؟ نقوم بتقليل عدد المحاولات المتبقية.

> لاحظ أيضاً أن الدالة `findLetter` تقوم بتحديث قيم الجدول `foundLetter` . تقوم بوضع 1 في الخانات الموافقة للحروف التي تمّ إيجادها.

الحلقة الرئيسية في اللعبة تتوقف هنا. لهذا فسنعيد من بداية الحلقة و نختبر ما إن كان قد بقي شيء من المحاولات للعب و اللاعب لم يربح بعد.

عند الخروج من الحلقة الرئيسيّة، لا يبقى سوى إظهار إن كان اللاعب قد نجح في اللعبة أو خسر قبل إنهاء البرنامج :

```
if (win(foundLetter))
        printf("\n\nYou win ! the secret word is : %s", secretWord );
else
        printf("\n\nYou lose ! the secret word is : %s", secretWord );
return 0;
}
```

سنستدعي الدالة `win` لنرى ما إن كان اللاعب قد ربح. إن كانت هذه هي الحالة، نقوم بإظهار الرسالة "ربح !"، و إلّا، فقد انتهت فرص اللعب، فقد خسر.

تحليل الدالة `win`

فلنرى الآن الشفرة الخاصة بالدالة `win` :

```
int win(int foundLetter[])
{
  int i = 0;
  int playerWins = 1;
  for (i = 0 ; i < 6 ; i++)
  {
    if (foundLetter[i] == 0)
        playerWins = 0;
  }
  return playerWins;
}
```

هذه الدالة تأخذ جدول القيم المنطقيّة `foundLetter` كمعامل. تعيد قيمة منطقية : "صحيح" إذا ربح اللاعب و "خطأ" إذا خسِر.

الشفرة الخاصة بهذه الدالة بسيطة، يفترض بك فهمها. نتصفّح `foundLetter` و نختبر ما إن كانت إحدى خانات الجدول تحوي "خطأ" (0). إن كان هناك حرف واحد لم يتمّ إيجاده فلقد خسر اللاعب : سيتم وضع "خطأ" (0) في المتغير المنطق `playerWins` . و إلّا، إن تمّ إيجاد كل الحروف، فالمتغيّر المنطقي سيكون "صحيحا" (1) و الدالة تعيد "صحيح".

تحليل الدالة `findLetter`

لهذه الدالة مهمّتان :

- إرجاع متغير منطقي يشير ما إن كان الحرف موجوداً في الكلمة السرية.
- تحديث (على 1) خانات الجدول `foundLetter` في المواضع الموافقة للحرف الذي تمّ إيجاده.

```
int findLetter(char letter, char secretWord[], int foundLetter[])
{
  int i = 0;
  int rightLetter = 0;
  // Search for the letter in the table foundLetter
  for (i = 0 ; secretWord[i] != '\0' ; i++)
  {
    if (letter == secretWord[i]) // If it exists
    {
      rightLetter = 1; // Memorize that it was the right one
      foundLetter[i] = 1; // Put the correspondent value to 1 in the table
    }
  }
  return rightLetter ;
}
```

نتصفّح إذن السلسلة المحرفيّة `secretWord` محرفاً محرفاً. في كلّ مرّة، نختبر ما إن كان الحرف الذي اقترحه اللاعب حرفا من الكلمة، سيتم القيام بأمرين :

- تعديل المتغير المنطقي `rightLetter`، إلى 1، لكي تعيد الدالّة 1 لأن الحرف متواجد بالفعل في `secretWord`.
- تحديث الجدول `foundLetter` على الموضع الحالي للإشارة إلى أنّ هذا الحرف قد تمّ إيجاده.

الشيء الجيّد في هذه الطريقة، هو أننا سنتصفّح كلّ الجدول (لا نتوقف عند أول حرف تم إيجاده). هذا سيسمح لنا بتحديث الجدول `foundLetter` بشكل صحيح، في الحالة التي تحتوي فيها الكلمة حرفاً مكرراً عدّة مرّات، مثل حالة L في YELLOW.

3.18 التصحيح (2 : استعمال قاموس الكلمات)

لقد قمنا بجولة حول الوظائف الأساسيّة لبرنامجنا. إنّه يحتوي على كلّ ماهو ضروري لإدارة جولة في اللعبة، لكنه لا يعرف كيف يختار كلمة عشوائية من قاموس كلمات. لم أضع لك شفرة المرحلة الأولى كلّها لأنها كانت ستأخذ حجماً كبيراً كما ستكون تكرارا مع الشفرة المصدريّة النهائيّة التي ستراها فيما بعد.

قبل الذهاب بعيدا، الشيء الأوّل الواجب فعله هو إنشاء قاموس الكلمات. و حتى إن كان قصيراً فهذا ليس سيّئا، سيكون مناسبا للاختبارات.

سأقوم إذن بإنشاء ملف `dico.txt` **في نفس دليل مشروعي**. حاليّا، سأضع فيه الكلمات التالية :

```
HOUSE
BLUE
AIRPLANE
XYLOPHONE
BEE
```

```
6 BUILDING
7 WEIGHT
8 SNOW
9 ZERO
```

ما إن أنتهي من كتابة البرنامج، سأعود بالطبع إلى هذا القاموس و أملؤه بالكثير من الكلمات الغريبة كـXYLOPHONE و المطوّلة كـANTIDISESTABLISHMENTARIANISM. لكن حاليّا، لنعد إلى كتابة التعليمات.

تحضير الملفات الجديدة

قراءة "القاموس" ستأخذ الكثير من الأسطر (على الأقل، لديّ إحساس قبليّ بذلك). لهذا فسآخذ الاحتياطات بإضافة ملف آخر إلى مشروعي `dico.c` (الذي سيتكفّل بقراءة القاموس). كما سنقُوم بإنشاء `dico.h` الذّي يحوي نماذج الدوال الموجودة في `dico.c`.

في `dico.c` سأبدأ بتضمين المكتبات التّي أنا في حاجة إليها بالإضافة إلى `dico.h`. أوّلا، كالعادة، سأحتاج إلى `stdio.h` و `stdlib.h` هنا. بالإضافة إلى هذا، يجب عليّ أن أقوم بسحب عشوائي لعدد من القاموس، سأقوم إذن بتضمين `time.h` مثلما فعلنا سابقاً من أجل العمل التطبيقي الأول "أكثر أو أقل".سأحتاج أيضا إلى تضمين `string.h` من أجل استعمال `strlen` في نهاية الدالّة.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#include "dico.h"
```

الدالة findWord

هذه الدالة تأخذ معاملا واحدا : مؤشراً نحو الذاكرة حيث يمكن كتابة الكلمة. هذا المؤشّر يتم تزويدنا به عن طريق `main`. الدالة ستعيد `int` و سيكون قيمة منطقية : 1 = تمّ كلّ شيء على مايرام، 0 = كان هناك خطأ ما.

هذه بداية الدالة :

```c
int findWord(char *chosenWord)
{
  FILE* dico = NULL; // The pointer of the file
  int wordsNumber = 0, chosenWordNumber = 0, i = 0;
  int readCharacter = 0;
```

أعرّف بعض المتغيرات التّي ستكون ضرورية لي. مثل الـ`main` ، لم يخطر ببالي وضعها كلّها من البداية، يوجد بالتأكيد من قمت بإضافتها لاحقا حينما عرفت أنني بحاجة إليها.

أسماء الكلمات تعبّر عن نفسها. لدينا المؤشّر على القاموس `dico` و الذي سيمكننا من قراءة `dico.txt`، متغيرات مؤقّتة ستخزّن المحارف، إلخ. لاحظ أنني هنا استعملت `int` لتخزين محرف (`characterRead`) لأنّ الدالة `fgetc` التّي سأستخدمها تعيد `int`. فمن الأفضل إذن تخزين النتيجة في `int`.

فلنمر إلى التالي :

```c
dico = fopen("dico.txt", "r"); // Open the dictionary in read mode only
// Check if it's open without a problem
if (dico == NULL) // If there's a problem
{
  printf("\nImpossible to load words dictionary");
  return 0; // Return a zero to say that the function failed
  // The function stops after reading the instruction return
}
```

ليس لديّ الكثير لأضيفه هنا. أفتح الملف `dico.txt` بوضع قراءة فقط (`"r"`) و أتأكّد إن نجحت عن طريق اختبار إذا كان `dico` يحمل القيمة `NULL` فإن عملية فتح الملف قد فشلت (ملفّ غير موجود أو مفتوح من طرف برنامج آخر). في هذه الحالة سنظهر رسالة خطأ و نقوم بـ `return 0`.

لماذا تضع `return` هنا ؟ في الحقيقة، التعليمة `return` تضع نهاية للدالة. إذا لم يتم فتح القاموس، فستتوقف الدالة و لن يذهب الحاسوب إلى أبعد من ذلك. إعادة 0 تشير للـ `main` أنّ الدالّة قد فشلت.

في ما يلي من الدالّة نفترض أن فتح الملف نجح.

```c
// Count the number of words in the file (Just counting the \n signs)
do
{
  characterRead = fgetc(dico);
  if (characterRead == '\n')
    wordsNumber++;
} while(characterRead != EOF);
```

هنا، نتصفّح كلّ الملف دفعة واحدة باستعمال `fgetc` (محرفا بمحرف). نعدّ الـ `\n` التي نجدها. أي أنه في كلّ مرة نلتقي بـ `\n` نزيد قيمة المتغير `wordsNumber`.

بفضل هذه الشفرة سنتحصل في المتغير `wordsNumber` على عدد الكلمات الموجودة في الملف. تذكّر بأن الملف يحتوي على كلمة في كل سطر.

```c
chosenWordNumber = aleatoryNumber(wordsNumber); // Take a word by hazard
```

هنا أستدعي دالّة من إنشائي تختار لي عددا عشوائيا بين 1 و `wordsNumber` (المعامل الذي ترسله للدالة).

إنها دالة بسيطة وضعتها أيضاً في الملف `dico.c` (سأشرحها بشكل موسّع لاحقاً). باختصار، تقوم بإرجاع عدد (يوافق رقم سطر الكلمة في الملف) عشوائي يتم تخزينه في `chosenWordNumber`.

```
// Start reading the file from the beginning and stop when finding the right word
rewind(dico);
while (chosenWordNumber > 0)
{
        characterRead = fgetc(dico);
        if (characterRead == '\n')
         chosenWordNumber --;
}
```

و الآن و نحن نملك رقم الكلمة التي سنختارها، سنعود إلى بداية الملف باستدعاء `rewind()` ، و سنتصفّح الملف محرفا بمحرف لنحسب عدد `\n` . هذه المرّة، سنقوم بانقاص قيمة `chosenWordNumber` . إن اخترنا مثلا الكلمة رقم 5، في كلّ ادخال سيتمّ إنقاص المتغير `chosenWordNumber` بواحد.

سيأخذ إذن القيم 4 ثم 3 ثم 2 ثم 1 ثم 0.

عندما يصل المتغير إلى 0، نخرج من الـ`while` ، لأن الشرط `chosenWordNumber > 0` لم يعد محققا.

هذا الجزء من الشفرة، و الذي يجب عليك فهمه حتما، سيريك كيفية تصفّح الملف للوصول إلى المكان المراد. الأمر ليس معقّدا و لكنه ليس "بديهيا" أيضاً. كن متأكّدا من فهم ما أقوم بفعله هنا.

الآن، يفترض أن نملك مؤشّرا متموضعا تماماً قبل الكلمة السريّة الّتي يجب إيجادها. سنقوم بتخزينها في `chosenWordNumber` (المعامل الذي تستقبله الدالة) بفضل `fgets` بسيط يقوم بقراءة الكلمة :

```
/* The cursor of the file is placed in the best place.
Nothing is needed more than an fgets that will read the line */
fgets(chosenWord, 100, dico);
// We erase the \n at the end of the word
chosenWord[strlen(chosenWord) - 1] = '\0';
```

نحن نطلب من `fgets` ألا تقرأ أكثر من 100 محرف (هذا هو حجم الجدول `chosenWord` ، الذي قمنا بتعريفه في الـ`main`). تذكّر أنّ `fgets` تقرأ سطرا كاملا، بما في ذلك `\n` . بما أننا لا نريد إبقاء الـ`\n` في الكلمة النهائية، نحذفها باستبدالها بـ`\0` . لهذا تأثير القيام بقطع الكلمة قبل `\n` .

و ها نحن ذا ! لقد خزّنا الكلمة السرية في الذاكرة عند عنوان `chosenWord` .

لم يتبقّ سوى غلق الملف، و إعادة القيمة 1 لتتوقف الدالة و تشير إلى أن كل شيء على ما يُرام :

```
  fclose(dico);
  return 1; // Everything is okay, return 1
}
```

انتهينا من الدالة `findWord` !

الدالة aleatoryNumber

هذه الدالة التي وعدتك بشرحها سابقاً. نختار عدداً عشوائيا و نعيده :

```
int aleatoryNumber(int maxNumber)
{
    srand(time(NULL));
    return (rand() % maxNumber);
}
```

السطر الأوّل يهيّئ مولّد القيم العشوائية، مثلما تعلّمنا فعل ذلك في العمل التطبيقي الأوّل "أكثر أو أقل". أما السطر الثاني فيقوم باختيار عدد عشوائي بين 0 و `maxNumber` و يعيده. يمكنك ملاحظة أنني قمت بكلّ ذلك في سطر واحد، هذا ممكن بكلّ تأكيد، رغم أنّه قد يبدو أحيانا أقل قابليّة للقراءة.

الملف dico.h

يحتوي نماذج الدوال فقط. يمكنك أن تلاحظ "الحماية" التي تقدّمها `#ifndef` التي كنت قد طلبت منك تضمينها في كلّ ملفاتك ذات الامتداد `.h` (راجع فصل توجيهات المعالج في حالة الحاجة) :

```
#ifndef DEF_DICO
#define DEF_DICO
int findWord(char *chosenWord);
int aleatoryNumber(int maxNumber);
#endif
```

الملف dico.c

هذا هو الملف `dico.c` كاملا :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "dico.h"
int findWord(char *chosenWord)
{
    FILE* dico = NULL; // The pointer on the file
    int wordsNumber = 0, chosenWordNumber = 0, i = 0;
    int characterRead = 0;
    dico = fopen("dico.txt", "r"); // Open the dictionary in read mode only
    // Check if it's open without a problem
    if (dico == NULL) // If there's a problem
    {
        printf("\nImpossible to load words dictionary");
        return 0; // Return a zero to say that the function failed
```

```
17          // The function stops after reading the instruction return
18      }
19      // Count the number of words in the file (just count the \n characters)
20      do
21      {
22          characterRead = fgetc(dico);
23          if (characterRead == '\n')
24              wordsNumber++;
25      }
26      chosenWordNumber = aleatoryNumber(wordsNumber); // Take a word by hazard
27      // Start reading the file from the beginning and stop after finding the
            right word
28      rewind(dico);
29      while (chosenWordNumber > 0)
30      {
31          characterRead = fgetc(dico);
32          if (characterRead == '\n')
33              chosenWordNumber--;
34      }
35      fgets(chosenWord, 100, dico);
36      // Erase the \n at the end of the word
37      chosenWord[strlen(chosenWord) - 1] = '\0';
38      fclose(dico);
39      return 1; // Everything is okay, return 1
40  }
41  int aleatoryNumber(int maxNumber)
42  {
43      srand(time(NULL));
44      return (rand() % maxNumber);
45  }
```

يجب إذن تعديل الـmain !

و الآن بما أن الملف `dico.c` جاهز، سنعود للدالة `main` كي نقوم بتحديثها على حسب التغييرات التي قمنا بإجرائها.

سنبدأ أوّلا بتضمين `dico.h` إذا أردنا استدعاء دوال الملف `dico.c`. بالإضافة إلى ذلك، سنقوم أيضاً بتضمين `string.h` لأننا سنستعمل الدالة `strlen` :

```
1  #include <string.h>
2  #include "dico.h"
```

للبدأ، سيتم تغيير كيفية تعريف المتغيرات، فنحن مثلاً لن نهيّئ قيمة المتغير `secretWord`، سننشئ فقط جدول محارف من `char` (100 خانة).

بالنسبة للجدول `foundLetter` فحجمه سيعتمد على طول الكلمة التي سنختارها من القاموس، و بما أننا لازلنا لا نعرف هذا الطول، سنكتفي بتعريف مؤشّر. لاحقاً سنستعمل الدالة `malloc` و جعل هذا المؤشّر يُؤشّر على الخانة التي

سيتم حجزها.

و هذا مثال يعبّر تماما عن حاجتنا الماسة لاستعمال الحجز الحيّ : نحن لا نعرف حجم الجدول قبل ترجمة الشفرة، أي أننا مجبرون على تعريف مؤشّر و استدعاء `malloc`.

لا يجب أن ننسى تحرير الذاكرة حين لا نحتاج إلى الخانة التي تم حجزها، و لهذا سيتم استعمال الدالة `free` في نهاية الـ`main`.

نحتاج أيضاً إلى متغير `wordSize` و الذي سيحتوي ... حجم الكلمة السرية. في الواقع، لو نلاحظ الـ`main` كما كان في الشفرة السابقة، فسنرى أنه كلّما احتجنا حجم الكلمة استعملنا 6 (لأن الكلمة كانت YELLOW ذات 6 حروف). لكن حاليّا، بما أن الكلمة ستتغير، فيجب على البرنامج أن يتلاءم مع كل الكلمات.

إليك إذن التعريفات النهائية للمتغيّرات في الدالة `main` :

```
1 int main(int argc, char* argv[])
2 {
3         char letter = 0; // Stores the letter suggested by the user
4         char secretWord[100] = {0}; // The word that the user must find
5         int *foundLetter = NULL; // Boolean table. Each box corresponds to a
             letter in the secret word. 0 = letter not found, 1 = letter found
6         int remainingTries = 10; // Counting the remaining tries  (0 = dead)
7         int i = 0; // A little variable to browse the table
8         int wordSize= 0;
```

ستتغير بداية الدالة `main`، فلنلاحظ هذا :

```
1 if (!findWord(chosenWord))
2         exit(0);
```

نحن نستدعي أولا الدالة `findWord`، و ذلك يتم مباشرة داخل الشرط `if`. الدالة `findWord` ستقوم بوضع الكلمة التي اختارتها من القاموس في المتغير `secretWord`. كما أنها ستقوم بإرجاع متغير منطقي لنا لتخبرنا ما إن كانت العملية ناجحة أم لا، أي أننا نقرأ الشرط كالتالي : إذا لم يعمل الأمر فسنوقف البرنامج (`exit(0)`).

```
1 wordSize = strlen(secretWord);
```

نقوم بتخزين طول `secretWord` في المتغير `wordSize` كما شرحتُ سابقاً.

```
1 foundLetter = malloc(wordSize  * sizeof(int)); // We allocate dynamically the
     table foundLetter (that we don't know its size in the beginning)
2 if (foundLetter  == NULL)
3         exit(0);
```

و الآن سنحجز مكاناً في الذاكرة للجدول `foundLetter`. سنقدّم له حجم الكلمة `wordSize`. سنختبر بعد ذلك ما إن كان المؤشّر يساوي `NULL`. إذا كان كذلك، فالحجز قد فشل. في هذه الحالة سنوقف البرنامج حالا (باستعمال `exit`).

إذا تمّت قراءة الأسطر السابقة، فكلّ شيء قد عمل تماما.

هذه هي أهم التعديلات على الـ`main`، يبقّى أن تقوم باستبدال كل تكرار للرقم 6 بالمتغير `wordSize`. مثال :

```
for (i = 0 ; i < wordSize ; i++)
        foundLetter[i] = 0;
```

هذه الشفرة تقوم بوضع القيمة 0 في كل خانة من الجدول `foundLetter`.

كان يفترض أن أضع نموذج الدالة `win` لأضيف المتغير `wordSize`. فبدون هذا لا يمكن للدالة معرفة متى توقف الحلقة التكرارية.

هذه هو الملف `main.c` كاملا:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "dico.h"
int win(int foundLetter[], int wordSize);
int findLetter(char letter, char secretWord[], int foundLetter[])
char readCharacter();
int main(int argc, char* argv[])
{
        char letter = 0; // Stores the letter suggested by the user
        char secretWord[100] = {0}; // The word that the user must find
        int *foundLetter = NULL; // Boolean table. Each box corresponds to a
            letter in the secret word. 0 = letter not found, 1 = letter found
        int remainingTries = 10; // Counting the remaining tries  (0 = dead)
        int i = 0; // A little variable to browse the table
        int wordSize = 0;
        printf("Welcome !\n\n");
        if (!findWord(chosenWord))
                exit(0);
        wordSize = strlen(secretWord);
        foundLetter = malloc(wordSize  * sizeof(int)); // We allocate
            dynamically the table foundLetter ( that we don't know its size in
            the beginning )

        if (foundLetter  == NULL)
                 exit(0);
        for (i = 0 ; i < wordSize; i++)
                 foundLetter [i] = 0;
        while (remainingTries > 0 && !win(foundLetter , wordSize))
        {
                printf("\n\nYou have %ld remaining tries", remainingTries);
                printf("\'nWhats the secret word? ");
                for (i = 0 ; i < wordSize; i++)
                {
                                if (foundLetter [i]) // If we have found the
                                    letter n° i
                                printf("%c", secretWord[i]); // We display it
                                else
```

```
36                          printf("□"); // Else, we display a □ for the
                               letters that are not found
37              }
38               printf("\nSuggest a letter : ");
39              letter = readCharacter();
40              // If it's not the right letter
41              if (!findLetter(letter , secretWord, foundLetter ))
42              {
43                              remainingTries−−; // We decrement by 1 the
                                   remaining tries
44              }
45          }
46          if (win(foundLetter , wordSize))
47                  printf("\n\nYou win ! The secret word is : %s",secretWord);
48          else
49                  printf("\n\nTou lose ! The secret word is : %s", secretWord);
50          free(foundLetter ); // We free the allocated memory
51          return 0;
52  }
53  char readCharacter()
54  {
55          char character = 0;
56          character = getchar(); // We read the first character
57          character = toupper(character); // We uppercase the character
58          // We read other characters until we reach \n (to erase them)
59          while (getchar() != '\n') ;
60          return character; // We return the first character that we read
61  }
62  int win(int foundLetter[], int wordSize)
63  {
64          int i = 0;
65          int playerWins = 1;
66          for (i = 0 ; i < wordSize ; i++)
67          {
68                  if (foundLetter[i] == 0)
69                                  playerWins = 0;
70          }
71          return playerWins;
72  }
73  int findLetter(char letter, char secretWord[], int foundLetter[])
74  {
75          int i = 0;
76          int rightLetter = 0;
77          // We search for the letter in the table foundLetter
78          for (i = 0 ; secretWord[i] != '\0' ; i++)
79          {
80                  if (letter == secretWord[i]) // If it exists
81                  {
82                                  rightLetter = 1; // We memorize that it was the
                                       right one
```

```
83                                  foundLetter[i] = 1; // We put the
                                      correspondent value to 1 in the table
84                      }
85              }
86              return rightLetter ;
87      }
```

أفكار للتحسين

تنزيل المشروع

للبدأ، أدعوكم لتنزيل المشروع عبر الرابط التالي:

(10 Ko) https://openclassrooms.com/uploads/fr/ftp/mateo21/pendu_siteduzero.zip

إذا كنت تعمل على الماك أو اللينكس، قم بحذف الملف `dico.txt` و أنشئ واحداً جديداً. على أي حال فالملفات يتم حفظها بشكل مختلف على الويندوز : لهذا فقد تظهر لك بعض المشاكل لو استعملت المشروع كما هو. تأكّد من وجود كل كلمة في سطر وحدها، و ارجع إلى السطر بعد كتابة الكلمة الأخيرة في الملف (ليتم حسابها في الشفرة).

هذا سيساعدك في تجريب كيف يعمل المشروع و ربما إضافة بعض التعديلات و التحسينات عليه، إلخ. من المستحسن أن تكون قد قمت بنفسك برمجة اللعبة دون الحاجة إلى تنزيل مشروعي كما كتبته أنا، مع ذلك أؤمن بأن هذا العمل كان صعبا بالنسبة للبعض من القراء.

ستجد في هذا الملف `.zip` ملفات من `.h` و `.c` كما تجد الملف `.cbp` الخاص بالمشروع. إنه مشروع تم إنشاؤه بالبيئة التطويرية Code::Blocks.
إن كنت تستعمل بيئة تطويرية أخرى، فلا داعي للقلق، قم بإنشاء مشروع بنفسك، و ضع فيه يدويا الملفات `.h` و `.c` المتواجدة في الملف `.zip` . ستجد أيضاً الملف التنفيذي (`.exe`) و القاموس (`dico.txt`).

تحسين الـPendu

إن مستوى شفرة اللعبة هذه، لابأس به، لدينا الآن لعبة تفتح ملفا و تأخذ منه كلمة عشوائية.

و لكن مع ذلك سأعطيك بعض الأفكار التي يمكنك إدراجها في اللعبة بهدف تحسينها :

- لحد الآن اللعبة تقترح علينا جولة واحدة، فسيكون من الأحسن أن نستعمل حلقة تكرارية تسمح بلعب جولة ثانية إذا كان اللاعب يريد ذلك !
- يمكنك أيضاً أن تجعل اللعبة تسمح بلعب لاعبين، الأول يدخل الكلمة السرية و الثاني يحاول تخمينها !
- هل ستتمكن من رسم رجل (باستعمال الـ`printf` فقط، نحن في الكونسول، تذكر) يقوم بالتفاعل مع اللاعب كأن يتأسف في حال ما إن أخطأ اللاعب في إيجاد الكلمة ؟

- يمكنك أن تطلب من اللاعب أن يختار صعوبة اللعبة، و حسب الصعوبة تغيّر في عدد المحاولات المسموحة له.

حاول الاستفادة من هذا العمل التطبيقي جيداً، و أعد الكرّة حتى لو قرأت الشفرة الخاصة بي، حاول تطويرها و تحسينها، أريد منك أن تستطيع لاحقاً برمجة لعبة الـPendu و عيناك مغمضتان !

هيّا، بالتوفيق !

الفصل 19

إدخال نصّ بشكل أكثر أمانا

إدخال النصوص في لغة الـC هي من أكثر الأمور حساسية. أنت تعرف الدالة `scanf` التي تعرّفنا عليها في الفصول الأولى. ستقول : و أيّ الأدوات ستكون أكثر سهولة و طبيعية منها ؟ لكن جهّز نفسك، بعد هذا الفصل ستقول عنها أي شيء باستثناء "بسيطة".

الذين سيستعملون برنامجك هم بطبيعة الحال بشر. فهناك منهم من يخطئ في كتابة شيء، بينما هناك من يتعمّدون إرباك برنامجك بمعلومات غير منتظرة. فإن طلبت من المستعمل : ما هو عُمرك ؟ من يضمن لك بأنه لن يجيبك بـ:" اسمي فلان و أنا من البلد فلان" ؟

الهدف من هذا الفصل هو تعريفك إلى بعض المشاكل التي يمكن أن نواجهها أثناء استعمالنا للدالة `scanf`، و تقديم دالة بديلة أكثر أماناً و هي `fgets`.

1.19 حدود الدالة scanf

هذه الدالة التي نستعملها جميعاً من الفصول الأولى في الكتاب، هي سلاح ذو حدين :

- سهلة الاستعمال حينما نكون في مستوى "مبتدئ" ، و لهذا السبب عرّفتك بها.
- لكن الطريقة التي تعمل بها معقّدة و يمكن أن تكون خطيرة في بعض الحالات.

ألا يبدو الأمر متناقضا ؟ فإن الدالة `scanf` سهلة الاستعمال و في نفس الوقت أكثر تعقيداً مما نتصور، سأريك الحدود التي يمكن لهذه الدالة أن تصل إليها و ذلك بتقديم مثالين واقعيين.

إدخال سلسلة محارف تحتوي على فراغات

لنفرض أننا طلبنا من المستعمل أن يقوم بإدخال سلسلة محارف في الكونسول، و هو يقوم بكتابة فراغ في سلسلته :

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
        char name[20] = {0};
        printf("What's your name ? ");
        scanf("%s", name);
        printf("Ah ! Your name is %s !\n\n", name);
        return 0;
}
```

```
What's your name ? Mathieu Nebra
Ah ! Your name is Mathieu !
```

؟

لماذا اختفت الكلمة "Nebra" ؟

ذلك لأن الدالة `scanf` تتوقف عن القراءة حينما تصل إلى فراغ، أو رجوع إلى السطر أو محرف جدولة (tabulation). يعني أنك غير قادر على قراءة سلسلة محرفيّة تحتوي على فراغات.

م

في الواقع، الكلمة "Nebra" لازالت مخزّنة في الذاكرة، في شيء نسميه بالمتغير المؤقّت (buffer)، المرة القادمة عندما نستدعي الدالة `scanf` فهي ستقوم بقراءة الكلمة "Nebra" وحدها الموجودة في المتغير المؤقّت.

يمكننا استعمال الدالة `scanf` بشكل يسمح لها بقراءة الفراغات، لكن الأمر معقّد جدّا. لمن يصرّ على ذلك، يمكنك إيجاد دروس مفصّلة على الويب، مثل الدرس الأجنبي المتوفّر على هذا الرابط :

http://xrenault.developpez.com/tutoriels/c/scanf/

إدخال سلسلة محارف طويلة للغاية

يوجد مشكل آخر، أكثر خطورة، و هو **تجاوز الذاكرة**.

في الشفرة التي رأيناها، يوجد السطر التالي :

```
char name[5] = {0};
```

ترى أنني قمت بحجز 5 خانات من أجل الجدول المسمّى `name` الذي هو من نوع `char`. يعني أننا قادرون على تخزين كلمة من 4 محارف، بينما الحرف الأخير فهو محجوز لعلامة نهاية السلسلة `\0`.
إذا نسيت كلّ هذا فراجع فصل السلاسل المحرفية.

المخطط التالي يمثل المكان الذي هو محجوز للكلمة التي عرّفناها :



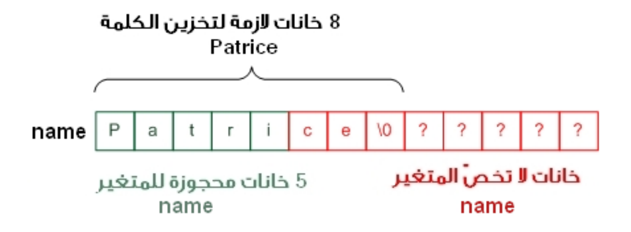
ماذا لو كتبنا عددا كبيرا من المحارف بالنسبة للمساحة المتوقّعة لتخزين المتغير ؟

```
What's your name ? Patrice
Ah ! Your name is Patrice !
```

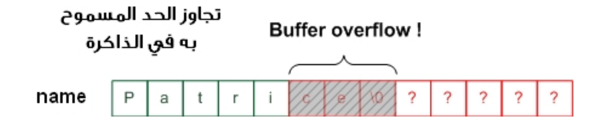
ستقول أن كل شيء على ما يرام لكن الواقع أنك بصدد مواجهة أكبر كابوس لدى المبرمجين !

لقد قمنا بـتجاوز في الذاكرة، هذا ما نسميه بـbuffer overflow بالإنجليزية.

كما ترى في المخطط التالي، لقد حجزت 5 خانات لكي تقوم باستعمال 8، ما الذي قامت به الدالة `scanf` ؟ لقد قامت بمواصلة الكتابة في الذاكرة وكأن شيئاً لم يحدث ! فلقد استغلّت خانات ليس لها الحق في الكتابة فيها.



الذي جرى في الحقيقة، هو أن المحارف الزائدة تسببت في مسح معلومات من الذاكرة و استبدالها بهذه المحارف. هذا ما نسميه بالـbuffer overflow.



لما الأمر خطير ؟

دون الدخول في التفاصيل، لأنه بإمكاننا البدء في محادثة قدر 50 صفحة و لا تتوقف أبداً، فلنقل بأنه إن لم يقم البرنامج بمعالجة حالات كهذه، فالمستعمل سيقوم بكتابة ما يحلو له و تخريب المعلومات المتواجدة في الخانات التالية من الذاكرة. أي أنه قادر على كتابة شفرة في تلك الخانات و برنامجك سيقوم بتشغيل تلك الشفرات و كأنها تابعة له، و هذا ما نسميه بالهجوم عبر المتغير المؤقت (Buffer overflow attack)، نوع من الهجومات المعروفة عند القراصنة، و لكنه صعب التحقيق.

إذا كنت مهتماً بهذا الموضوع، يمكنك قراءة المقال التالي من ويكيبيديا (حذار، إنّه مع ذلك معقد جدا) :

http://fr.wikipedia.org/wiki/D%C3%A9passement_de_tampon

الهدف من هذا الفصل هو تأمين قراءة البيانات و ذلك بمنع المستعمل من تجاوز الذاكرة و إحداث buffer overflow.
بالطبع كان بإمكاننا تعريف جدول كبير للغاية (10,000 خانة) لكن هذا لا يحلّ المشكل فالشخص الذي يريد الوصول إلى الذاكرة ما عليه سوى إدخال سلسلة يتجاوز طولها 10,000 محرف و سيعمل هجومه كما يريد.

الشيء المحزن هو أن معظم المبرمجين لا ينتبهون دائما لهذه الأخطاء، و لو أنهم قاموا بكتابة الشفرة من المرة الأولى بشكل نظيف و صحيح، لما ظهرت كثير من الثغرات التي نتحدّث عنها اليوم.

2.19 استرجاع سلسلة محارف

توجد العديد من الدوال القياسيّة في لغة C التي تسمح باسترجاع سلسلة نصّيّة. إضافة إلى الدالة `scanf` ، و التي من الصعب دراستها هنا، لدينا :

- `gets` : دالة تقرأ سلسلة محرفيّة كاملة لكنها خطيرة جدّا لأنها لا تعالج مشكل الـbuffer overflow.
- `fgets` : تشبه الدالة `gets` لكنها تحمي البرنامج و ذلك بالتحكم في عدد المحارف المكتوبة في الذاكرة.

أعتقد أن الأمر مفهوم : على الرغم من أنها دالّة قياسيّة في الـC، `gets` هي دالّة خطيرة جدّا. كل البرامج التّي تستخدمها عرضة لأن يكونوا ضحايا الـbuffer overflow.

سنرى كيف تعمل الدالة `fgets` ، و كيف نستعملها في برامجنا الخاصّة في مكان الدالة `scanf` .

الدالّة fgets

نموذج هذه الدالة، المتواجد في المكتبة `stdio.h` هو :

```
char *fgets(char *str, int num, FILE *stream);
```

من المهم أن نفهم هذا النموذج. معاملات الدالة هي التالية :

- `str` : مؤشّر نحو جدول في الذاكرة، أين ستتمكن الدالة من كتابة النص المدخل من طرف المستخدم.
- `num` : حجم الجدول `str` المرسل كمعامل أوّل.
لاحظ أنه لو قمت بحجز جدول من `char` ، فإن الدالة `fgets` ستقرأ 9 محارف على الأكثر (آخر خانة محجوزة للمحرف `\0` المشير إلى نهاية السلسلة).
- `stream` : مؤشّر نحو الملف الذي سنقرأ منه. في حالتنا "الملف المراد قراءته" هو الإدخال القياسي (Standard input)، أي لوحة المفاتيح. لطلب قراءة الإدخال القياسي نرسل المؤشّر `stdin` المعرّف تلقائيّا في الملفات الرأسيّة للمكتبة القياسيّة لـC ليشير إلى لوحة المفاتيح. مع ذلك، يمكن استخدام `fgets` لقراءة الملفّات، كما رأينا في الفصل الخاص بالملفّات.

الدالة ستقوم بإرجاع نفس المؤشّر `str` للإشارة إلى إن كانت القراءة قد تمت بشكل صحيح أم لا. يكفي إذا أن نختبر ما إن كانت قيمة هذا المؤشّر تساوي `NULL`، فإن كانت كذلك، فهناك خطأ.

فلنجرّب !

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
        char name[10];
        printf("What's your name ? ");
        fgets(name, 10, stdin);
        printf("Ah ! Your name is %s !\n\n", name);
        return 0;
}
```

```
What's your name ? NEBRA
Ah ! Your name is NEBRA
!
```

الدالة تعمل بشكل جيد، مع تفصيل بسيط : عندما تضغط على زر الإدخال، تقوم `fgets` بالاحتفاظ بـ`\n` الموافق، هذا ما يفسّر الرجوع إلى السطر بعد الكلمة "NEBRA" كما يظهر في الكونسول.

لا يمكننا أن نمنع هذه الدالة من كتابة المحرف `\n` لأن الدالة تعمل هكذا. بالمقابل، هذا لا يمنع كتابتنا لدالة خاصّة بالإدخال تقوم نفسها باستدعاء `fgets` و حذف ذلك المحرف !

كتابة دالتك الخاصّة بالإدخال باستخدام fgets

ليس صعباً جدّا أن نقوم بكتابة دالة خاصة بك تقوم ببعض التصحيحات من أجلك في كلّ مرّة.

سنسمّي هذه الدالّة `read` . ستقوم بإرجاع القيمة 0 إن كان هناك خطأ و 1 إن لم يكن.

حذف الرجوع إلى السطر \n

الدالة `read` تستدعي `fgets`، إذا تمّ كلّ شيء على ما يرام، ستبحث عن المحرف `\n` بمساعدة الدالة `strchr` الّتي يفترض بك معرفتها. إذا تمّ العثور على `\n`، فستستبدله بـ`\0` (نهاية السلسلة) لتجنّب الاحتفاظ بـ"علامة الإدخال".

هاهي الشفرة علّق عليها خطوة بخطوة :

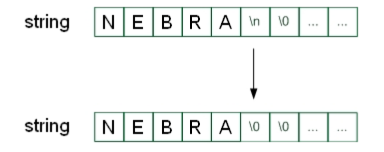
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // Think to include string.h for strchr()
int read(char *string, int length)
{
```

```
6         char *enterPosition = NULL;
7         // We read the text
8         if (fgets(string, length, stdin) != NULL)  // No error ?
9         {
10                enterPosition = strchr(string, '\n'); // We search for the "
                      Enter"
11                if (enterPosition != NULL) // If we find the \n
12                {
13                        *enterPosition = '\0'; // We replace the
                              character by \0
14                }
15                return 1; // We return 1 if there's no error
16        }
17        else
18         {
19                return 0; // We return 0 if there's error
20         }
21 }
```

ستلاحظ أنه بالإمكان استدعاء الدالة `fgets` مباشرة داخل `if`. هذا اختصار كتابي، كي لا أستعمل مؤشّرا يستقبل القيمة المُرجعة من الدالة ثم أختبر قيمته إن كانت `NULL` أم لا.

انطلاقا من `if` الأول، أعرف هل `fgets` عملت على ما يرام أم حدث مشكل ما (قام المستخدم بادخال محارف أكبر من العدد المسموح به).

إذا تم كل شيء بشكل جيد، سأذهب للبحث عن الـ `\n` باستعمال الدالة `strchr` ثم استبداله بالمحرف `\0` كما في الشكل التالي.



هذا المخطط يبيّن أن السلسلة الّتي تمت قراءتها من طرف الدالة `fgets` هي "`NEBRA\n\0`"، ثم قمنا باستبدال الـ `\n` بـ `\0` و هذا ما أعطانا "`NEBRA\0\0`".
ليس مشكلا أن توجد علامتا `\0` متتابعتين. الحاسوب سيتوقف عند الإشارة الأولى و يعتبرها نهاية السلسلة.

و النتيجة ؟ حسنا، لقد عملت بشكل جيّد.

```
1  int main(int argc, char *argv[])
2  {
3          char name[10];
4          printf("What's your name ? ");
5          read(name, 10);
6          printf("Ah ! Your name is %s !\n\n", name);
7          return 0;
8  }
```

```
What’s your name ? NEBRA
Ah ! Your name is NEBRA !
```

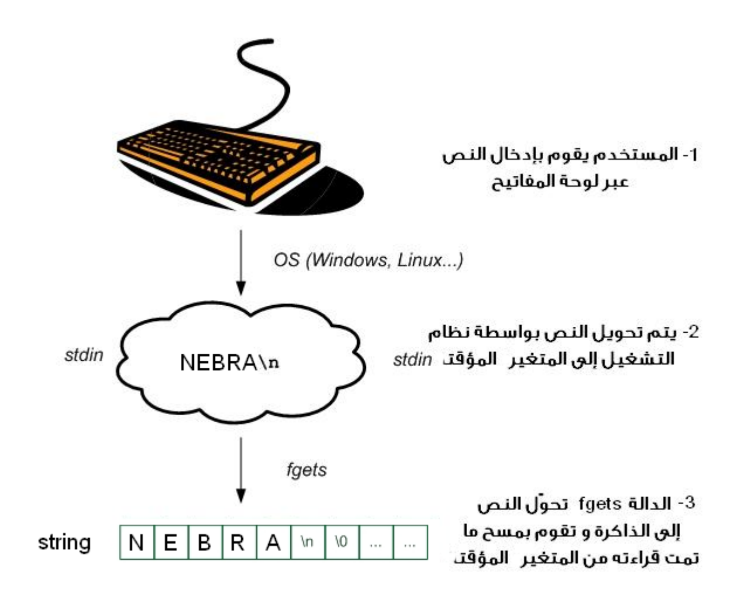
تفريغ المتغيّر المؤقّت

لم نصل بعد إلى نهاية الأمور المزعجة.
نحن لم نقم بتحليل الحالة التي يقوم فيها المستعمل بإدخال محارف أكثر من ما هو مسموح به !

```
What’s your name ? Jean Edouard Albert 1er
Ah ! Your name is Jean Edou !
```

بما أن الدالة `fgets` تدعم الحماية، فهي توقفت عند الحرف التاسع الذي قام المستعمل بإدخاله لأنّنا حجزنا جدولا من 10 محارف (يجب عدم نسيان أنّ العاشر محجوز لإشارة نهاية السلسلة). المشكل هو أن بقيّة السلسلة الّتي لم تتم قراءتها. "ard Albert 1er" لم تختف ! و إنّما لازالت موجودة في المتغير المؤقّت. هذا المتغير المؤقت هو مكان في الذاكرة يعمل كوسيط بين لوحة المفاتيح و الجدول الّذي سيتم تخزين السلسلة فيه. في الـC، لدينا مؤشّر نحو المتغير المؤقت، و هو `stdin` الذي تكلمنا عنه قبل قليل.

أعتقد أن مخططا صغيرا سيساعد على توضيح الأمور.



حينما يقوم المستعمل بإدخال نص بلوحة المفاتيح، فإن نظام التشغيل (Windows مثلا) يقوم بنسخ النص مباشرة في المتغير المؤقّت `stdin`.

مهمة الدالة `fgets` هي إحضار المحارف الموجودة في المتغير المؤقت ووضعها في الذاكرة التي قمت أنت بتحديدها (الجدول `string`).
بعد القيام بعملها، تمسح ما قامت بنسخه من المتغير المؤقّت.

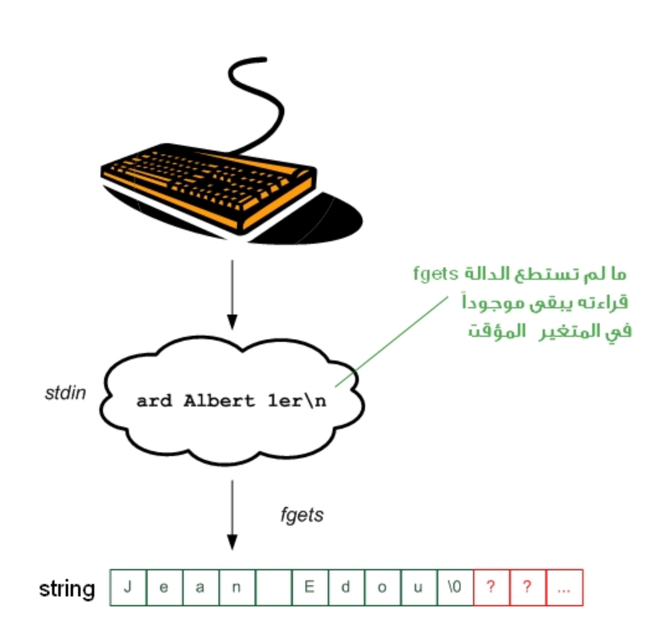
إذا عمل كل شيء على ما يرام، فالدالة `fgets` ستقوم بإفراغ كل محتوى المتغير المؤقّت، أي أن هذا الأخير سيكون فارغاً بعد استدعاء الدالة. لكن في الحالة التي يقوم المستخدم بإدخال كثير من المحارف لا يمكن أن يسعها المكان المحجوز لها، فإنه يتم مسح الحروف التي تمت قراءتها فقط، و بالتالي بعد استدعاء الدالة `fgets` فإن المتغير المؤقّت يحتوي دائما المحارف المتبقّية !

فلنجرّب مع سلسلة كبيرة :

```
1  int main(int argc, char *argv[])
2  {
3          char name[10];
4          printf("What's your name ? ");
5          read(name, 10);
6          printf("Ah ! Your name is %s !\n\n", name);
7          return 0;
8  }
```

```
What's your name ? Jean Edouard Albert 1er
Ah ! Your name is Jean Edou !
```

الدالة `fgets` قامت بنسخ المحارف التسعة الأولى كما كان متوقّعا. المشكل هو أن المحارف المتبقية لازالت في المتغير المؤقّت !



هذا يعني أنه لو استدعينا الدالة `fgets` مرة أخرى فإنها ستقوم بقراءة ما كان متبقّيا في المتغير المؤقّت !

فلنجرّب هذه الشفرة :

```
1  int main(int argc, char *argv[])
2  {
3          char name[10];
4          printf("What's your name ? ");
```

```
5        read(name, 10);
6        printf("Ah ! Your name is %s !\n\n", name);
7        read(name, 10);
8        printf("Ah ! Your name is %s !\n\n", name);
9        return 0;
10 }
```

نحن نقوم باستدعاء الدالة `read` مرّتين لكنك ستلاحظ أنّه لن يتم السماح لك بإدخال اسمك مرتين، وذلك لأن الدالة `fgets` لن تطلب من المستخدم إدخال أيّ نصّ في المرّة الثانية لأنّها ستجده في المتغير المؤقت !

```
What's your name ? Jean Edouard Albert 1er
Ah ! Your name is Jean Edou !
Ah ! Your name is ard Alber !
```

إذا قام المستعمل بإدخال محارف كثيرة، فإن الدالة `fgets` ستحمي البرنامج من مشكل تجاوز الذاكرة، لكن يبقى دائما آثار النصّ في المتغير المؤقت. لذا يجب تفريغ هذا الأخير.

سنقوم إذا بتحسين عمل الدالة `read`، و سنقوم في الحالات التي تتطلب ذلك باستدعاء دالة نسميها `clearBuffer`، لكي تتأكد من تفريغ المتغير المؤقت في حال ما احتوى على محارف زائدة :

```
1  void clearBuffer()
2  {
3          int c = 0;
4          while (c != '\n' && c != EOF)
5          {
6                  c = getchar();
7          }
8  }
9  int read(char *string, int length)
10 {
11         char *enterPosition = NULL;
12         if (fgets(string, length, stdin) != NULL)
13         {
14                 enterPosition = strchr(string, '\n');
15                 if (enterPosition != NULL)
16                 {
17                         *enterPosition = '\0';
18                 }
19                 else
20                 {
21                         clearBuffer();
22                 }
23                 return 1;
24         }
25         else
26         {
27                 clearBuffer();
28                 return 0;
```

```
29            }
30    }
```

الدالة `read` تستدعي الدالة `clearBuffer` في حالتين :

- السلسلة المدخلة طويلة جداً (يمكننا أن نعرف ذلك بعدم وجود الإشارة `\0` في السلسلة المنسوخة).
- إذا حدث أي خطأ مهما كان، يجب تفريغ محتوى المتغير المؤقت لأسباب حماية لكي لا يبقى شيء هناك.

الدالة `clearBuffer` قصيرة لكنها عميقة. فهي تقرأ المتغير المؤقت محرفاً محرفاً باستعمال الدالة `getchar` . هذه الدالة تقوم بإرجاع `int` (و ليس `char` ، سوف تعرف السبب لاحقا، أيّا يكن).
سنكتفي نحن باسترجاع القيمة في متغير `c` من نوع `int` . نقوم بحلقة تكرارية مادمنا لم نقرأ بعد المحرف `\0` أو الرمز `EOF` (نهاية الملف) و هما يعنيان : لقد وصلت إلى نهاية المتغير المؤقت. سنتوقّف عند الوصول إلى أحد هذين المحرفين.

يبدو عمل الدالة `clearBuffer` صعباً قليلاً لكنها تقوم بعملها. لا تتردد في تكرار قراءة الشرح عدة مرات من أجل الفهم الجيد.

3.19 تحويل سلسلة محرفيّة إلى عدد

الدالة `read` هي فعالة و قويّة الآن، لكنها تجيد قراءة النصوص فقط. ستتساءل حتما : "لكن كيف نقوم باسترجاع عدد ؟"

الحقيقة أن الدالة `fgets` هي دالة مبدئية. مع `fgets` لا يمكن قراءة سوى النصوص، و لكن توجد دوال أخرى تقوم بتحويل النص إلى عدد.

strtol : تحويل سلسلة محرفيّة إلى long

نموذج هذه الدالة خاص نوعاً ما :

```
1 long strtol(const char *start, char **end, int base);
```

الدالة ستقوم بقراءة السلسلة المحرفيّة المرسلة إليها (`start`) و ستحاول تحويلها إلى `long` باستعمال الأساس المحدد (`base`) (غالباً ما نستعمل الأساس 10، لأننا نستعمل الأرقام من 0 إلى 9، و لهذا ضع مكانه العدد 10).
ستقوم بإرجاع العدد الذي نجحت في قراءته.
بالنسبة لمؤشر المؤشر `end` ، فالدالة ستقوم باستغلاله لإرجاع أول محرف صادفته و لم يكن رقماً. لكننا لسنا بحاجة إليه، فلنكتفي بوضع `NULL` مكانه لنقول أننا لا نريد استرجاعه.

على السلسلة المحرفيّة أن تبدأ برقم، فبعد الأرقام كل شيء يتم تجاهله. يمكن أن تكون مسبوقة بفراغات.
هذه أمثلة للفهم الجيد :

```
long i;
i = strtol("148", NULL, 10); // i = 148
i = strtol("148.215", NULL, 10); // i = 148
i = strtol(" 148.215", NULL, 10); // i = 148
i = strtol(" 148+34", NULL, 10); // i = 148
i = strtol(" 148 dead leaves", NULL, 10); // i = 148
i = strtol( " There are 148 dead leaves", NULL, 10 ); // i = 0 (error : The
    string doesn't start with a number)
```

كل السلاسل التي تبدأ برقم (أو ربما بفراغات قبله) سيتم تحويلها إلى `long` حتى الوصول إلى محرف غير مقبول (نقطة، فاصلة، علامة استفهام، زائد، إلخ).

بالنسبة لسلسلة لا تبدأ بأرقام و أو بفراغات تليها أرقام، فلا يمكن تحويلها و بالتالي تقوم الدالة بإرجاع القيمة 0.

يمكننا كتابة الدالة `readLong`، و التي تقوم باستدعاء الدالة `read` (لقراءة النص)، و بعد ذلك تحويل النص إلى عدد :

```
long readLong()
{
        char textNumber[100] = {0}; // 100 cells are sufficient
        if (read(textNumber, 100))
        {
                // If we read the text without problems, we convert textNumber
                   to long and we return it
                 return strtol(textNumber, NULL, 10);
        }
        else
        {
                 // If there's a problem, we return 0
                return 0;
         }
}
```

يمكنك تجريب الشفرة داخل `main` بسيط.

```
int main(int argc, char *argv[])
{
        long age = 0;
        printf("How old are you ? ");
        age = readLong();
        printf("Ah ! You are %d years old !\n\n", age);
        return 0;
}
```

```
How old are you ? 18
Ah ! You are 18 years old!
```

strtod تحويل سلسلة محرفيّة إلى double

الدالة `strtod` مطابقة للدالة `strtol`، الفرق الوحيد هو أنها ستحاول قراءة عدد عشريّ و إرجاع `double`.

```
double strtod(const char *start, char **end);
```

تجد أن المعامل الثالث `base` اختفى هنا، بينما يبقى مؤشّر المؤشّر `end` الّذي لا يفيدنا في شيء.

على خلاف الدالة السابقة، فإن هذه الدالة ستأخذ في الحسبان "النقطة" العشرية. عندما أقول نقطة يعني أن الدالة لا تقبل الفاصلة "," (يبدو أنّها مبرمجة من طرف ناطقين بالإنجليزية).

فلتقم بكتابة الدالة `readDouble` بنفسك. إن كتابتها مماثلة للدالة `readLong`، الاختلاف الوحيد هو أنها ستستدعي الدالة `strtod` ثم ستقوم بإرجاع قيمة `double`.

يعني أنك ستكتب التالي في الكونسول :

```
What's your weight ? 67.4
Ah ! Your weight is 67.400000 kg !
```

حاول بعد ذلك تعديل الدالة `readDouble` لتقبل الفاصلة أيضاً كفاصل عشري. إن الأمر بسيط : فقط قم باستبدال كل تكرار للمحرف "`,`" بالمحرف "`.`" (بالاستعانة بالدالة `strchr`)، ثم قم ببعث النص الجديد إلى الدالة `strtod`.

ملخّص

- الدالة `scanf` بالرغم من أنها تبدو سهلة و طبيعية إلا أنها معقّدة و تفرض علينا بعض الحدود. فمثلاً، هي لا تقبل قراءة نص يحتوي فراغات.
- نقول أننا تسببنا في الـbuffer overflow إذا تجاوزنا المساحة المخصصة في الذاكرة، فمثلاً لو قام المستخدم بإدخال 10 محارف و نحن قد حجزنا 5 خانات فقط.
- الحل الأمثل هو استدعاء الدالة `fgets` لتقوم باسترجاع النص الذي يُدخله المستعمل.
- يجب أن تجتنب استعمال الدالة `gets` بأي ثمن لأنّها لا تحمي من الـbuffer overflow.
- يمكنك كتابة دالة خاصة بك، تقوم باستدعاء الدالة `fgets` كما فعلنا لكيّ تحسّن عملها.

الجزء ج

إنشاء ألعاب 2D في SDL

الفصل 20

تثبيت الـSDL

ابتداءً من الآن، إنتهت الدروس النظرية ! لأننا سنمرّ إلى مرحلة مهمّة، و سنستمتع بالتطبيق بالاستعانة بمكتبة نسميها SDL.

في الفصول السابقة كنّا قد تطرّقنا تقريباً لكلّ أساسيات اللغة C، لكن تبقى هناك دائماً بعض التفاصيل الصعبة نوعاً ما لنكتشفها. سأقول لك بأنه يُمكن لهذا الكتاب أن يتوقّف هنا مخبرا إيّاك : "نعم لقد تعلّمت البرمجة بلغة C"، لكني متأكّد بأن الجميع سيشاركني الرأي لو قلت بأن المُبرمج سيحسّ نفسه دائماً مبتدئاً مادام لم "يخرج" من الكونسول !

الـSDL هي مكتبة تُستخدم خاصّة لإنشاء ألعاب ثنائية الأبعاد. سنتعرّف في هذا الفصل على هذه المكتبة و نتعلّم كيف نقوم بتثبيتها.

نسمي هذا النوع من المكتبات بمكتبات الطرف الثالث (Third party libraries). يجب أن تعرف أنه يوجد نوعان من المكتبات :

- **المكتبة القياسية** (Standard library) : و هي المكتبة القاعدية التي تعمل على كلّ أنظمة التشغيل (من هنا تم استنباط الكلمة standard) و هي تسمح بالقيام بأمور بسيطة كـ `printf`. هذه المكتبات يتمّ تسطيبها تلقائيّا عند تثبيتك للبيئة التطويرية و المترجم.

  خلال الجزئين الأوّلين من هذا الكتاب، كنّا قد استعملنا المكتبة القياسيّة فقط (`stdlib.h`، `stdio.h`، `string.h`، `time.h` ...). لم نقم بدراستها بالتفصيل لكنّا جرّبنا منها جزءً كبيراً. إن كنت تريد معرفة المزيد عن هذا النوع من المكتبات أجْرِ بحثاً في Google، مثلاً بكتابة "C standard library"، و ستجد نماذج الدوال في هذه المكتبة، بالإضافة إلى شرح قصير حول دور كلّ دالة.

- **مكتبات الطرف الثالث** (Third party libraries): هي مكتبات لا يتم تثبيتها تلقائيا. و إنّما يجب عليك تنزيلها من الأنترنت و تثبيتها بنفسك على حاسوبك.

  على عكس المكتبات القياسية، التي تكون بسيطة نسبيّا و تحتوي على عدد قليل من الدوال، فإنه توجد الآلاف من مكتبات الطرف الثالث، و التي تمت كتابتها من طرف مبرمجين آخرين. بعضها جيّدة، و أخرى أقل، بعضها مدفوع، و بعضها الآخر مجاني، إلخ. الأمر المثالي هو إيجاد مكتبة جيّدة و مجانية في نفس الوقت !

إنه لمن المستحيل أن أضع لك درساً يشرح كل المكتبات الموجودة. حتّى لو أمضيت حياتي كلّها 24 ساعة / 24، لن أستطيع !
لذا سأقدّم لك مكتبة واحدة فقط مكتوبة بالـC و مُستعملة من طرف مبرمجين مثلك.

هذه المكتبة تدعى *SDL*. السؤال المطروح هو لماذا اخترت هذه المكتبة بالضبط ؟ ما الذي يميّزها عن باقي المكتبات ؟ هذه أسئلة سأبدأ في الإجابة عليها إنطلاقاً من الآن.

1.20 لماذا نختار الـSDL ؟

اختيار مكتبة ليس بالأمر السهل !

كما قلت لك الآن، توجد الآلاف من المكتبات للتنزيل.
بعضها بسيط، و بعضها كبير جداً لدرجة أن درساً كهذا لا يكفي أن يشرحها كلّها !

الاختيار صعب. لكنّي اخترت هذه المكتبة، التي هي نوعاً ما سهلة الاستعمال، كبداية. ستكون هذه إذا أوّل مكتبة تقوم باستعمالها (إذا لم نحسب المكتبة القياسية).

إنه من الواضح أن أغلب القرّاء يريدون معرفة كيفية فتح نوافذ، إنشاء لعبة، إلخ. و لكن إن كنت تحب الكونسول فيمكننا الاستمرار فيها لوقت أطول، إذا أردت، لا ؟ إذا لدينا هنا بعض الفضول !
أودّ كثيراً أن أريك كيف تعمل كلّ هذه الأمور، لكننا سنحاول أن نتطرّق إليها خطوة بخطوة، و بالنسبة للأعمال التطبيقية، فلدينا عملان تطبيقيان لهذا الجزء من الكتاب !

لقد اخترت لك مكتبة سهلة و قوية، ستكون كبداية لك في تحقيق (تقريبا) أحلامك المتعلّقة بالواجهة الرسومية، و من دون تعب (حسناً، كلّ شيء نسبيّ بالطبع !).

الـSDL، اختيار جيّد !

سنقوم الآن بدراسة هذه المكتبة. لماذا اخترتها هي و ليس أخرى ؟



- **هي مكتبة مكتوبة بلغة C** : أي أنه بإمكان المبرمجين أن يستعملوها في برامجهم المكتوبة بالـC. و كما هو الحال بالنسبة لأغلب المكتبات المكتوبة بالـC، يمكن استعمالها في لغة الـ++C بالإضافة إلى لغات برمجية أخرى.
- **هي مكتبة حُرّة و مجانية** : و هذا كي لا تضطرّ لدفع أي ثمن مقابل استعمالك ما سأقدّمه لك في بقيّة الكتاب. على عكس ما قد نعتقد، إيجاد مكتبة جيدة و مجانية ليس أمراً صعباً كثيراً، فقد انتشرت كثيراً في أيامنا هذه. المكتبة الحرة هي ببساطة مكتبة يمكنك الحصول على الشفرة المصدرية الخاصة بها. في حالتنا هذه، رؤية الشفرة ليس مُهمّا

بالنسبة لنا. لكن كونها حرة يفتح لنا الباب من أجل ميزات أخرى أهمّها المداومة (أي أنه إن توقف صاحب المكتبة عن تطويرها، يُمكن لمبرمجين آخرين أن يكملوا عمله)، بالإضافة إلى مجّانيّتها غالبا. هذا يعني عدم إمكانيّة اختفاء المكتبة في يوم من الأيام.

- **يُمكنك إنشاء برامج تجارية ذات ملكية خاصة بفضل هذه المكتبة.** قد أكون قد تسرّعت بهذا الكلام، لكنّه يجب اختيار مكتبة حرّة تمنحك الحريّة الأقصى. الحقيقة أنه يوجد نوعان من المكتبات الحُرة :
  - المكتبات تحت رخصة **GPL** : مكتبات مجانية، و يمكنك رؤية الشفرة المصدرية الخاصة بها، لكن بشرط أن تقوم أنت كذلك بنشر الشفرة المصدرية الخاصة بالبرنامج الذي أنشأته باستخدامها.
  - المكتبات تحت رخصة **LGPL** : مثل سابقتها، لكن ليس عليك أن تنشر الشفرة المصدرية الخاصة بالبرنامج. أي أنه يمكنك بها إنشاء برامج مملوكة.

> بالرغم من أنه يمكنك قانونيّا عدم نشر الشفرة المصدرية الخاصة بالبرنامج، إلا أنني أنصحك بذلك. فبهذا يمكنك أن تأخذ رأي المبرمجين الأكثر تمرّساً منك. و هذا يسمح لك بالتحسّن. بعد هذا، فإن إنشاء برنامج حُر أو ذو ملكية خاصة، يرجع لطبيعة تفكير كل شخص. لن أدخل في نقاش بخصوص هذا الموضوع، لكن فلتعلم أن كلّ النوعين له ميزاته و مساوءه.

- هي مكتبة **متعددة المنصّات** (Multi-platform) : سواء كنت على Windows، Mac OS X أو GNU/Linux، ستعمل لديك هذه المكتبة. و الحقيقة أن هذه نقطة قوّة يراها المبرمجون بالمكتبة : يمكنها أن تعمل على عدد كبير جداً من أنظمة التشغيل، فعلى غرار Windows، Mac OS X و GNU/Linux، فهي تشتغل أيضاً على Atari، Amiga، Symbian، Dreamcast ... إلخ. أي أنه بالإمكان لبرامجك أن تعمل حتى على أجهزة Atari القديمة ! مع ذلك يجب القيام ببعض التعديلات و ربّما استخدام مترجم خاص. لن أدخل في التفاصيل هنا.
- أخيرا، فإن هذه المكتبة تسمح لك بالقيام بالكثير من **الأمور الممتعة** التي سنتعرّف إليها من خلال الفصول القادمة. لا أقول أنّ مكتبة رياضيّاتية قادرة على حلّ معادلات من الدرجة الرابعة ليست ممتعة، لكنّي سأركّز على أن يكون هذا الدرس سهلا قدر الإمكان لكيّ يحثّك على البرمجة.

هذه المكتبة ليست مخصصة فقط لإنشاء ألعاب الفيديو. سأعترف بأن معظم البرامج التي تمت كتابتها بهذه المكتبة، هي عبارة عن ألعاب، لكن هذا لا يعني أنك مجبر لاستعمالها من أجل ذلك. كما نعلم، كلّ شيء ممكن بالعمل و الاجتهاد. كنت قد رأيت من قبل محرر نصوص تمت برمجته بالـSDL، على الرغم من أنّه هناك مكتبات أخرى أحسن لهذا الغرض. إن كنت تريد برمجة واجهة رسومية تقليديّة تسمح بإظهار نافذة، زر، قائمة، إلخ. فأنا أنصحك إذا بالتوجّه إلى المكتبة GTK+.

الإمكانيات المتاحة بالـSDL

المكتبة SDL هي مكتبة منخفضة المستوى. هل تتذكر أول الكتاب حينما حدّثتك عن لغات البرمجة عالية المستوى و لغات البرمجة منخفضة المستوى ؟ هذا ينطبق على المكتبات أيضاً.

- **المكتبات منخفضة المستوى** : تحتوي على دوال قاعدية جدّا. يوجد عدد قليل من هذه الدوال لأنّه يمكننا القيام بكلّ شيء بها. و هذه الدوال لبساطتها تكون سريعة جدّا. لهذا فالبرامج المنشأة بهذا النوع من المكتبات تكون عادة الأسرع.
- **المكتبات عالية المستوى** : تحتوي على الكثير من الدوال التي تسمح بالقيام بالكثير من المهام. هذا يجعلها أبسط من ناحية الاستخدام.

  لكن هذا النوع من المكتبات يكون عادة "كبيرا"، و ليس من السهل دراستها و معرفتها بأكملها. كما أنها قد تكون أثقل من المكتبات منخفضة المستوى (لكنّ هذا قد لا يكون واضحا).

على العموم، لا يمكننا القول بأن "مكتبة منخفضة المستوى هي أحسن من مكتبة عالية المستوى" أو العكس. فكلّ منهما لها مميزات و مساوئ. الـSDL التي سنقوم بدراستها، تنتمي إلى المكتبات منخفضة المستوى.

يجب إذا أن تتذكر بأن الـSDL تقدّم دوالا قاعدية. يمكنك إذا الرسم بيكسلا بيبكسل، رسم مستطيل أو إظهار صور. هذا كلّ شيء، و صدّقني أنّ هذا كافٍ.

- بتحريك صورة، يمكنك أن تقوم بتحريك شخصيّة.
- بإظهار العديد من الصور الواحدة تلو الأخرى بسرعة، يمكنك إنشاء تحريك (Animation).
- بوضع العديد من الصور، الواحدة بجنب الأخرى، يكون باستطاعتك إنشاء لعبة حقيقيّة.

كمثال عن لعبة تم صنعها بالـSDL، اعلم أنّ اللعبة الشهيرة "Civilisation : Call to power"، تم دعمها في نظام اللينكس لاحقاً باستخدام الـSDL.



يجب أن تعلم أنّ جودة اللعبة تعود إليك و إلى الفريق الذي تعمل معه. إن كان لديك مصمم موهوب، فيمكنك صنع لعبة أجمل.

الشيء الوحيد الذي يحدّ الـSDL هو أنها تقتصر على الألعاب ثنائية الأبعاد، و لم تُنشأ من أجل الألعاب ثلاثية الأبعاد. هذه أمثلة على ألعاب يمكن تحقيقها بالـSDL (ليست سوى قائمة صغيرة، كلّ شيء ممكن مادام ثنائيّ الأبعاد) :

- Breakout
- Bomberman
- Tetris
- ألعاب المنصّات : Super Mario Bros، Sonic، Rayman، ...
- RPG ثنائية الأبعاد : Zelda، الأجزاء الأولى للعبة Final Fantasy، إلخ.

لا يمكن وضع لائحة كاملة، الأمر يعود فقط للقدرة على التخيّل. و صدّقني بأنك قادر على برمجة ألعاب فائقة الروعة. فلقد رأيت أحد القرّاء ينشئ تهجينا بين Breakout و Tetris.

فلنعد إلى الأرض و نمسك خيط هذا الفصل. سنقوم الآن بتسطيب المكتبة، لنتمكّن من التقدّم في العمل.

2.20 تنزيل الـSDL

الموقع الرسمي للمكتبة SDL سيصبح قريبا الوجهة الّتي نقصدها كثيرا. هناك، يوجد كلّ ما تحتاجه، بدءً من المكتبة نفسها و مرورا إلى التوثيق (Documentation) الخاص بها.

http://www.libsdl.org/

إذهب إلى اللائحة Download المتواجدة على يسار الصفحة الرئيسية للموقع.
اختر النسخة الأحدث الّتي تجدها (SDL 1.2 عندما كتبت هذه السطور).

صفحة التنزيل مجزّأة إلى عدة أجزاء.

- **الشفرة المصدرية** (Source code) : هنا يمكنك تحميل الشفرة المصدرية الخاصة بالمكتبة. أدري أن القراء فضوليون ليعرفوا كيف تعمل المكتبة من الداخل، لكنّ هذا لن يفيدنا. الأسوأ هو أنّه سيقوم بإلهائك عن هدفنا الرئيسي.
- **مكتبات وقت التشغيل** (Runtime libraries) : هي الملفات التي تحتاج إلى تقديمها مع الملف التنفيذي حين تريد أن تعطي برنامجك لشخص آخر. بالنسبة للويندوز، أنا أتكلم عن الملف `SDL.dll` . هذا الأخير يجدر به أن يتواجد إما :
  - بنفس المجلّد الذي يحتوي الملف التنفيذي (أنا أنصحك بهذا). الأحسن دائما هو أن تعطي الـDLL مع الملف التنفيذي و تبقيهم في نفس المجلد. إذا وضعت الـDLL في المجلّد الخاص بالويندوز، لن يكون عليك إلحاق الـDLL مع كلّ مجلّد يحتوي البرنامج SDL. و مع ذلك قد تحدث بعض المشاكل في حال ما قمت بمسح نسخة أحدث من الـDLL.
  - في المجلّد `C:\Windows` .
- **مكتبات التطوير** (Development libraries) : هي الملفات `.a` (أو `.lib` بالنسبة للVisual) و الملفات `.h` التي تسمح بإنشاء برامجك SDL. هذه الملفات ليست مفيدة إلا بالنسبة إليك أنت فقط المبرمج. أي أنه ليس عليك تقديمها مع ملفات البرنامج حين تنتهي من هذا الأخير.

إذا كنت تعمل في الويندوز، فسأعطيك ثلاثة نسخ، و ذلك حسب المترجم الخاص بك :

- VC6 : بالنسبة للذين يستخدمون النسخ القديمة غير المجانية من Visual studio (لا أعتقد أن هناك من القرّاء من لا زال يستعمل هذه النسخ). ستجد فيها على أي حال الملفات `.lib`.
- VC8 : بالنسبة للذين يستعملون Visual Studio 2005 Express أو نسخة أحدث، ستجد فيها الملفات `.lib`.
- mingw32 : بالنسبة للذين يستعملون Code::Blocks (ستجدون فيها إذا الملفات `.a`).

الشيء الخاص هنا، هو أن "مكتبات التطوير" تحتوي كل الملفات `.h` و الملفات `.a` (أو `.lib`) بالطبع، لكنها تحتوي أيضاً الملف `SDL.dll` و ملفات التوثيق الخاصة بالـSDL !
باختصار، كلّ ما عليك تنزيله هو "مكتبات التطوير"، فكلّ ما تحتاجه يتواجد بداخلها.

X

لا تخطئ في الرابط ! قم باختيار الـSDL في قسم "Development libraries" و ليس من قسم "Source code" !

؟

ما هو التوثيق (Documentation) ؟

التوثيق هو قائمة تحوي اللائحة الكاملة للدوال الخاصة بمكتبة معيّنة. و كل هذه الملفات تكون مكتوبة بالانجليزية (حتى لو كان كاتبوها مبرمجين فرنسيين). هذا سبب آخر يدفعك للتقدّم في لغة Shakespeare !

محتوى ملفات التوثيق ليس عبارة عن درس، بل هي عادة موجزة. الشيء الإيجابي بالنسبة لدرس، هي أنّها تحتوي قائمة لكل الدوال الموجودة، فهي إذا المرجع للمبرمج.
في كثير من الأحيان ستجد مكتبات بدون دروس تشرح كيفية عملها. و هنا لا يبقى لك سوى التوثيق الّذي نسميه عادة "doc" و يجب عليك تدبّر أمرك بهذا فقط (حتّى لو كان هذا صعبا أحيانا عندما تبدأ من دون أيّة مساعدة). المبرمج الحقيقيّ هو من يتمكّن من إيجاد ضالّته في الـ"doc".

لحدّ الآن، أنت لست بحاجة إلى التوثيق الخاص بالـSDL لأنني أنا من سيشرح لك كيفية عملها. لكن بما أنني غير قادر على أن أشرح لك كل الدوال التي بها، ستحتاج إلى قراءة التوثيق لاحقا.

ملفّات التوثيق توجد أصلاً في الحزمة "Development libraries" كما سبق و ذكرت، لكن بإمكانك تنزيلها وحدها من القائمة `Documentation` / `Downloadable`.
أنصحك أن تجمع ملفات HTML الخاصة بالتوثيق في مجلّد خاصّ (اسمه مثلاً `Doc SDL`) ثم إنشاء اختصار إلى الفهرس `index.html`. و الهدف من هذا هو الوصول إلى هذه الملفات بشكل أسرع حينما تحتاج إليها.

3.20 إنشاء مشروع SDL : Windows

تثبيت مكتبة قد يكون أكثر صعوبة قليلاً مما تعوّد عليه الجميع. هنا لا يوجد تثبيت تلقائيّ يطلب منك أن تنقر "التالي"، "التالي"، "التالي"، "إنتهى".

الحقيقة أن تثبيت مكتبة أمر صعب على المبتدئين. لكن لأقوم برفع المعنويات فإن تسطيب مكتبة SDL أمر سهل جداً مقارنة بتسطيب مكتبات أخرى أُتيحت لي فرصة استخدامها من قبل (هناك من يتم إعطاؤك منها الشفرة المصدرية فقط، بينما أنت تتولى أمر الترجمة !).

و الحقيقة أن كلمة "تثبيت" ليست الملائمة هنا. لن نقوم بتثبيت أي شيء، فقط نريد أن نصل إلى الكيفية التي ننشئ فيها مشروع SDL في البيئة التطويرية الخاصة بنا.
ستختلف كيفية التعامل حسب البيئة التطويرية التي تستعملها. سأقوم بتقديم الطريقة الخاصة بكلّ بيئة من بيئات التطوير التي قدّمتها في بداية الكتاب، و هكذا كي يستطيع الجميع المتابعة.

سأعرض الآن كيف ننشئ مشروع SDL في كلّ واحد من البيئات الثلاث السابقة.

تسطيب الـSDL في Code::Blocks

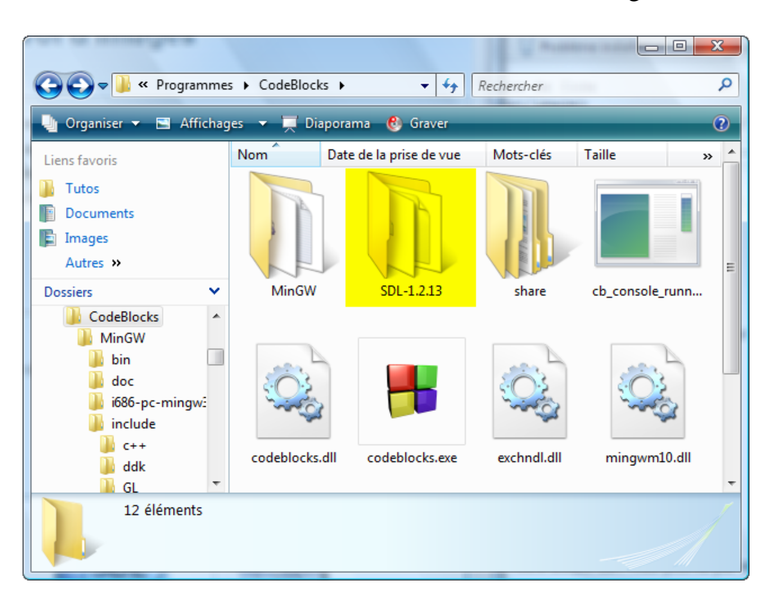
استخراج ملفات الـSDL

افتح الملف المضغوط "Development Libraries" الذي قمت بتنزيله.
هذا الملف هو بامتداد `.zip` بالنسبة للـVisual و `.tar.gz` بالنسبة للـmingw32 (يلزمك برنامج مثل Winrar أو 7-Zip لكي تقوم بفك الضغط عن الملفات ذات الصيغة `.tar.gz`).

الملف المضغوط يحتوي العديد من المجلّدات الداخلية، و هذه هي الملفات التي تهمّنا :

- `bin` : يحتوي الملف `.dll` الخاص بالـSDL.
- `docs` : يحتوي الملفات التوثيقيّة الخاصة بالـSDL.
- `include` : يحتوي الملفات الرأسية `.h`.
- `lib` : يحتوي الملفات `.lib` (أو `.a` بالنسبة لـCode::Blocks)

يجب عليك استخراج كل الملفات و المجلّدات الداخلية ووضعها في مكان ما بالقرص الصلب لحاسوبك، يمكنك مثلا وضعها في مجلد خاص بـSDL داخل مجلّد الـCode::Blocks.

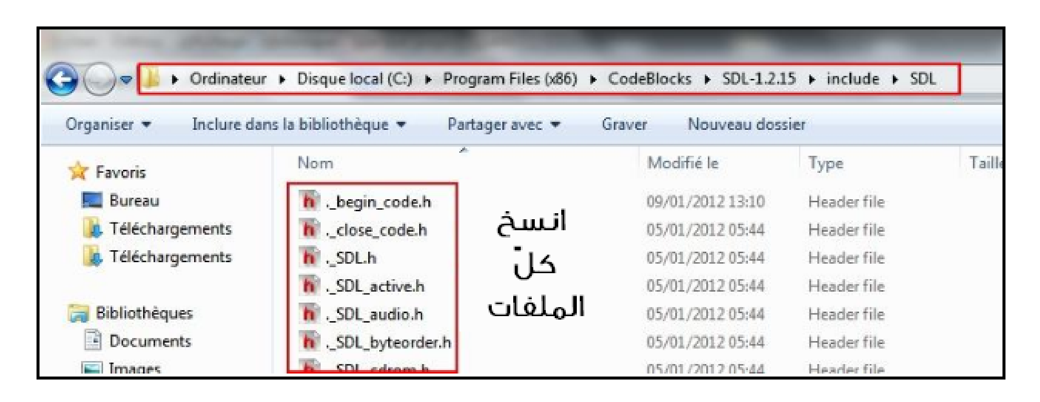


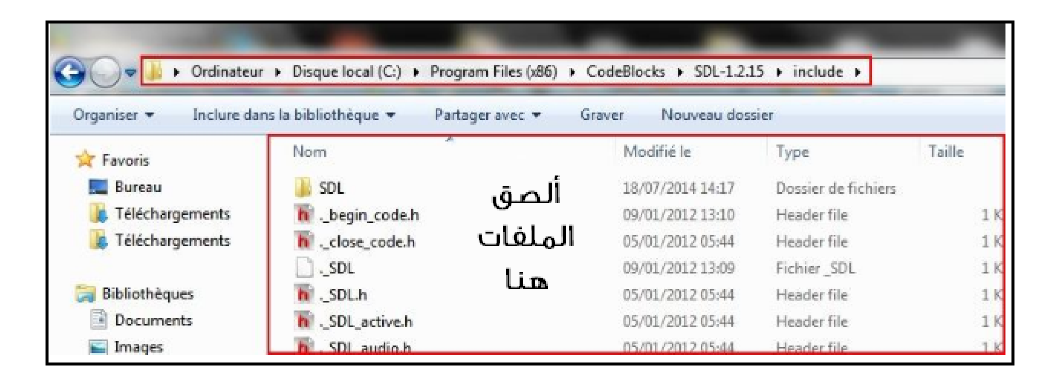
بالنسبة لي فإن المسار هو التالي :

```
C:\Program Files (x86)\CodeBlocks\SDL-1.2.13
```

احفظ المسار الذي به البرنامج، ستحتاج إليه عندما تريد تعديل إعدادات Code::Blocks لاحقا.

و الآن، علينا بالقيام بخطوة بسيطة، لتسهيل الأمور علينا، توجّه إلى المسار `include/SDL` (في حالتي، هو متواجد بـ `C:\Program Files (x86)\CodeBlocks\SDL-1.2.13\include\SDL`)، قم بنسخ الملفات الرأسية `.h` في المجلّد الأب، (أي في : `C:\Program Files (x86)\CodeBlocks\SDL-1.2.13\include`).

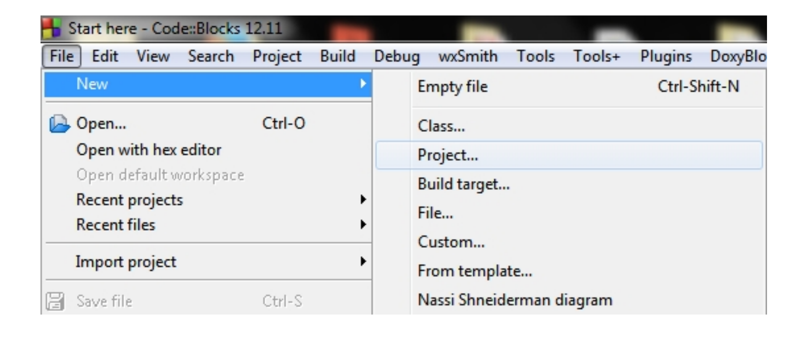




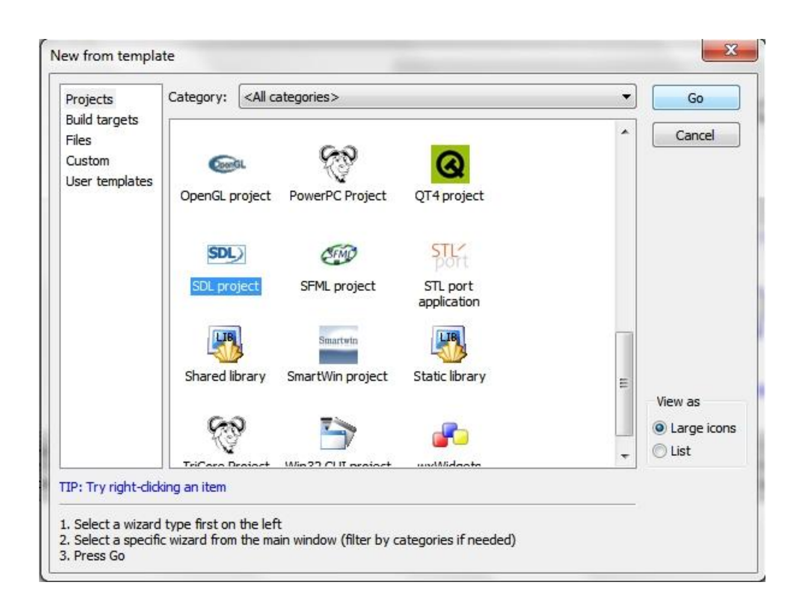
ها قد تم تسطيب المكتبة، فلنقم الآن بتعديل إعدادات Code::Blocks.

إنشاء مشروع SDL

افتح Code::Blocks و قم بإنشاء مشروع جديد.



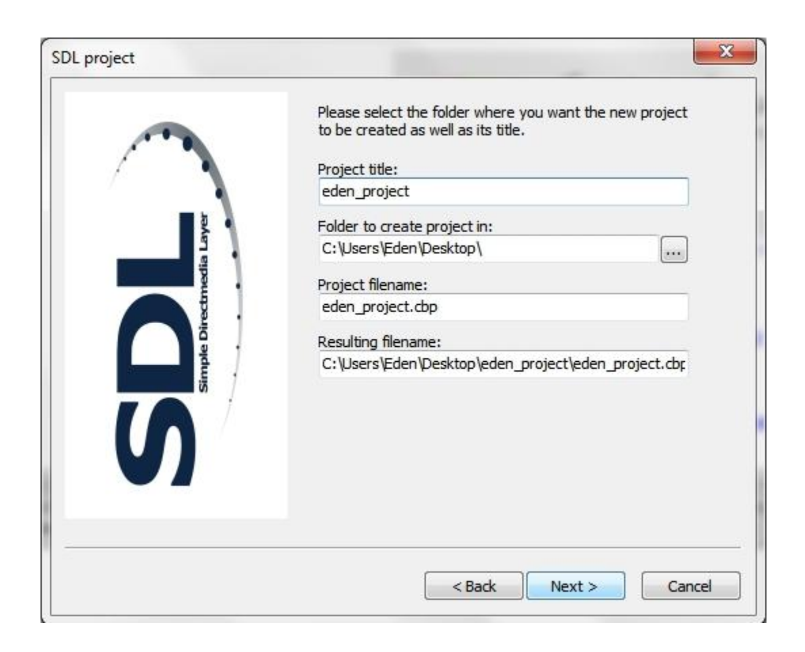
عوض أن تقوم باختيار Console Application كما جرت العادة، اختر SDL project.



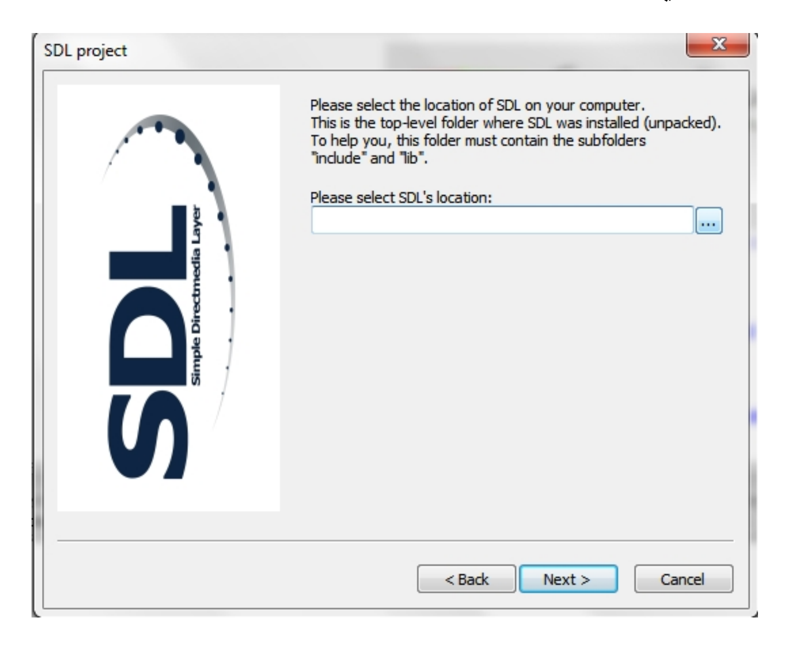
النافذة الأولى لا جدوى منها، قم بتجاوزها بالضغط على "التالي" (Next).



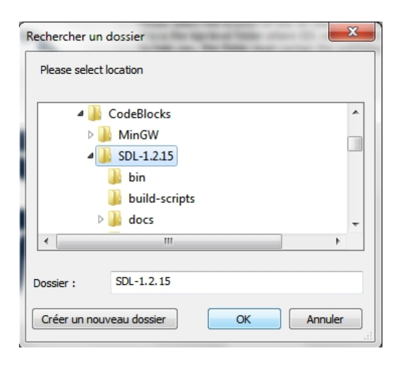
سيُطلب منك أن تقوم بإدخال اسم المشروع، قم بذلك كالعادة :



الآن يجب اختيار المسار الذي ثبتنا فيه المكتبة :

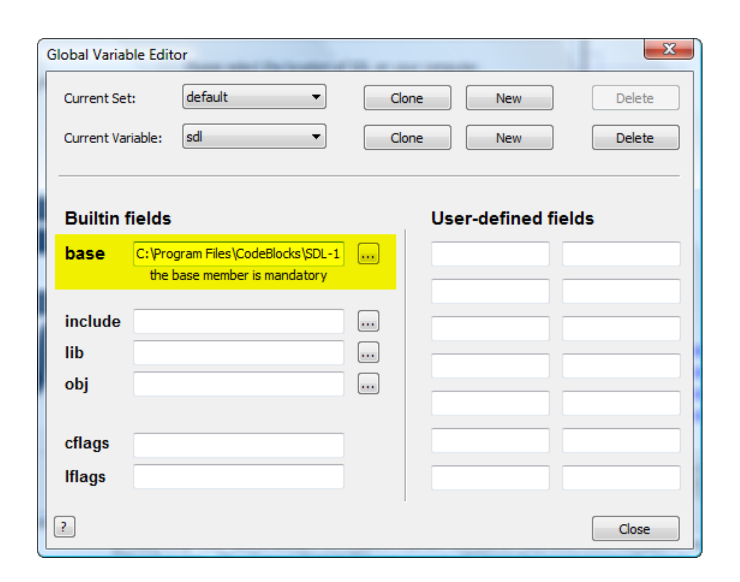


اضغط على الزر الذي يأخذ شكل مربّع به ثلاث نقاط، ستظهر لك النافذة التالية :

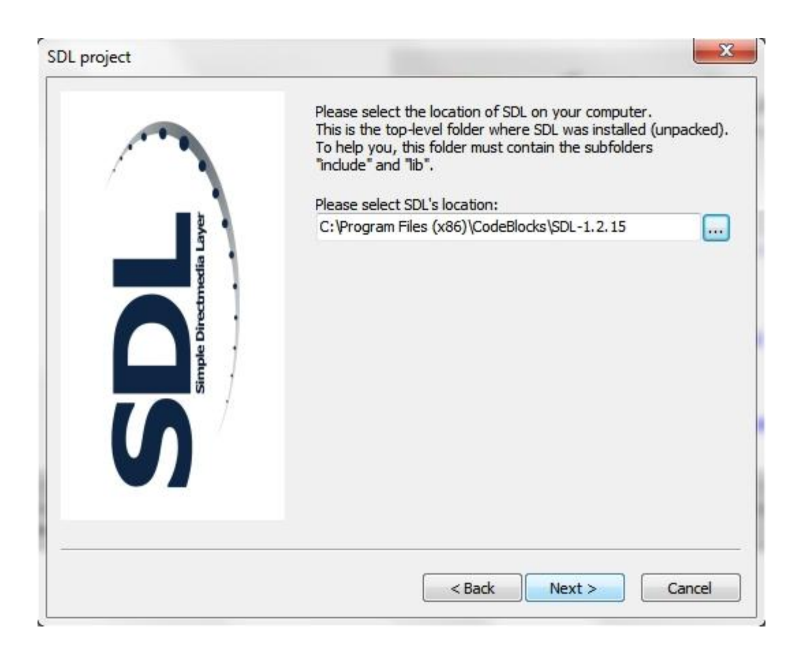


قم باختيار المسار (بالنسبة لي هو `C:\Program Files (x86)\CodeBlocks\SDL-1.2.13`).

قد تظهر لك في مكان النافذة السابقة هذه النافذة :



املأ الحقل base بنفس الطريقة السابقة، ثمّ اضغط على زر الخروج، و ستلاحظ أن المسار قد تم تسجيله كالتالي :



اضغط على Next، ستظهر لك نافذة اختيار المترجم، قم باختيار الأوضاع Realease أو Debug (هذا لا يهم).

أخيرا اضغط على "إنهاء" (Finish). سيتم إنشاء المشروع التجريبي :

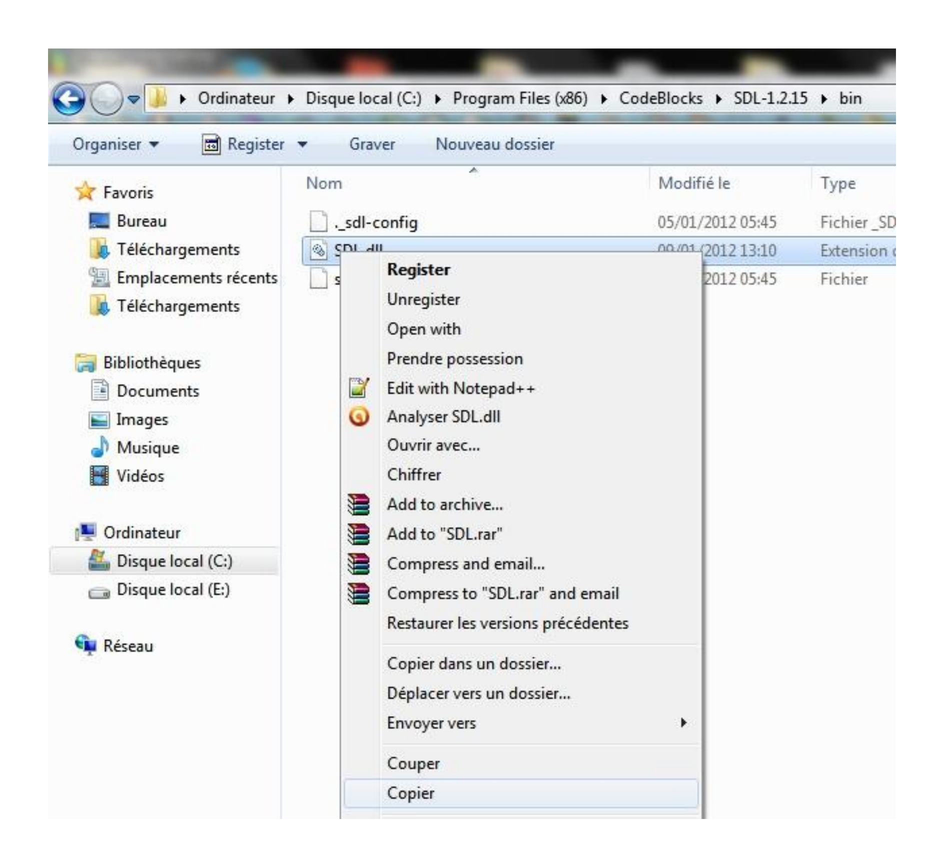
```
#ifdef __cplusplus
    #include <cstdlib>
#else
    #include <stdlib.h>
#endif

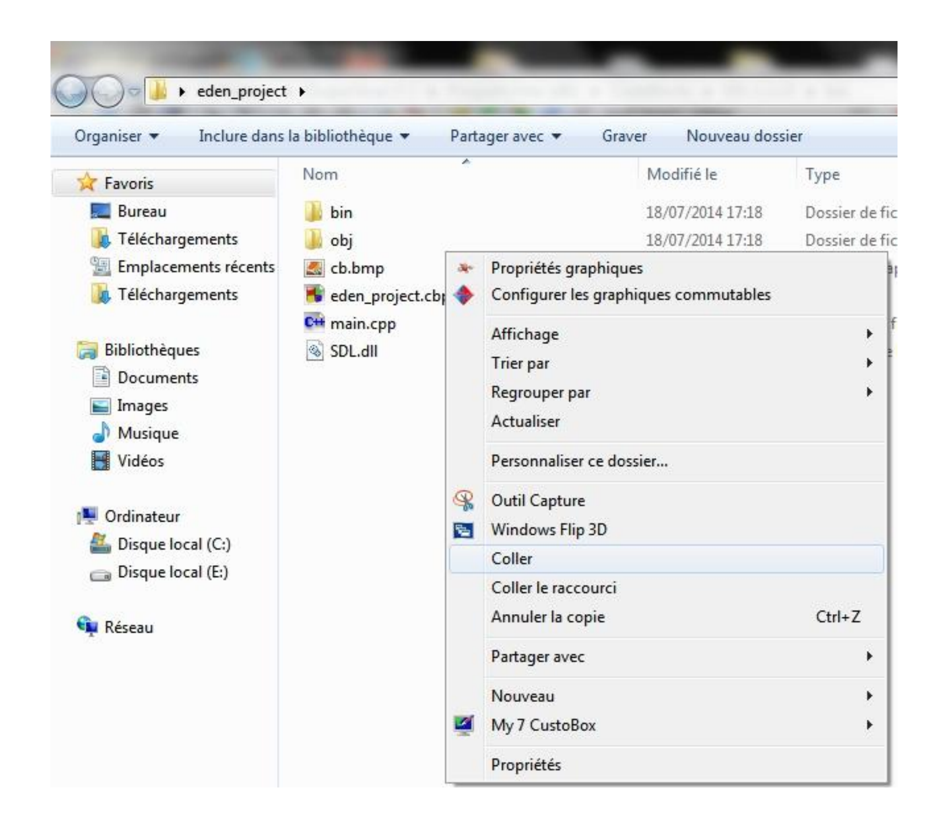
#include <SDL/SDL.h>

int main ( int argc, char** argv )
{
    // initialize SDL video
    if ( SDL_Init( SDL_INIT_VIDEO ) < 0 )
    {
        printf( "Unable to init SDL: %s\n", SDL_GetError() );
        return 1;
    }

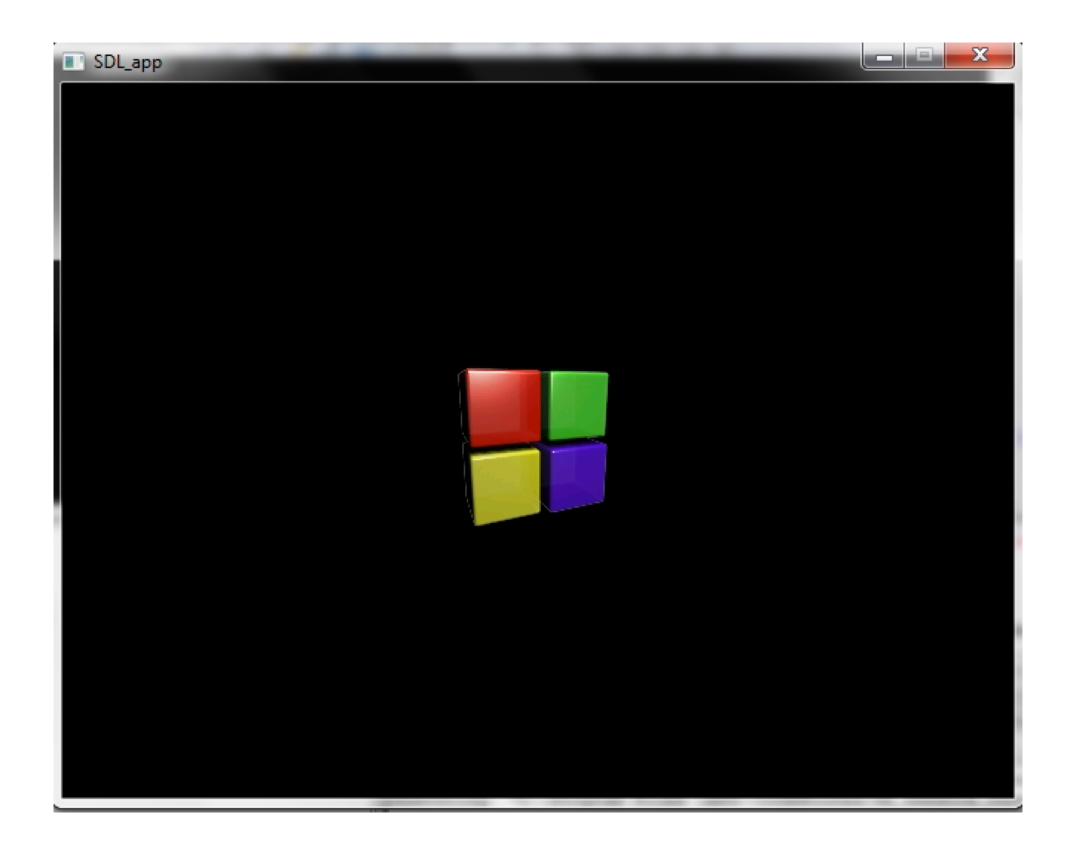
    // make sure SDL cleans up before exit
```

يحتوي المشروع على ملفين `main.cpp` و ملف `.bmp` قبل أن تحاول القيام بالترجمة. يجب القيام بخطوة أخيرة (عليك القيام بها دائما)، و هي نسخ الملف `SDL.dll` من ملفات المكتبة (الّذي يفترض أن يكون في المسار `C:\Program Files (x86)\CodeBlocks\SDL-1.2.13\bin\SDL.dll`) و ضعه في المجلّد الخاص بالمشروع.





أخرج من Code::Blocks و أعد الدخول إليه، ثم قم بترجمة البرنامج المُقترح مسبقاً. يفترض أن تظهر النافذة التالية :



إذا ظهرت لك النافذة السابقة، فهنيئا لك، المكتبة مثبتة بشكل جيد !

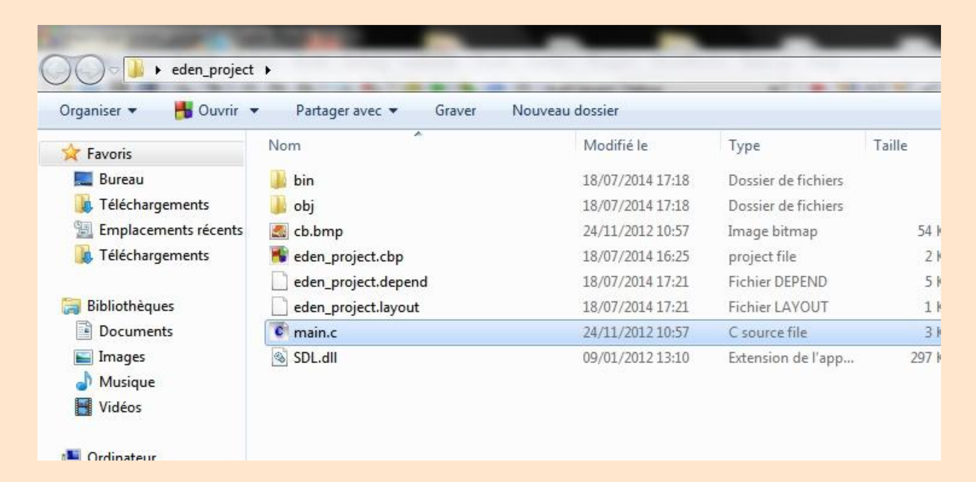
م

إن ظهرت لك الرسالة "The application can't start because the file SDL.dll is missing" أي أنه لا يمكن تشغيل البرنامج، لأن ملف `SDL.dll` غير موجود، فهذا يعني أنك لم تقم بنسخ الملف الأخير في ملفات المشروع كما طلبت منك !
و كما قلت، إن كنت تريد تسليم المشروع إلى أصدقائك، عليك بارفاق الملف التنفيذي `.exe` بالملف `SDL.dll`، بينما أنت لست بحاجة إلى إعطائهم الملفات `.h` و `.a` التي لا تهم أحدا سواك.

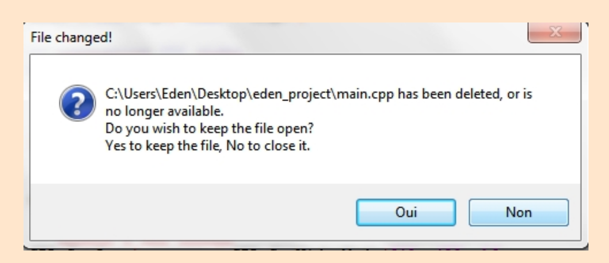
ملاحظات مترجمة الكتاب

كلّ الشرح الموجود في هذا الجزء يعطي الطريقة التي استخدمتها مترجمة الكتاب لتشغيل البرنامج في حالة ما لم تعمل الطريقة السابقة (الأصليّة). هذا يعني أنّ هذه الفقرات التالية ليست موجودة في الكتاب الفرنسي الأصلي، و إنّما هي مساهمة شخصيّة من المترجمة.

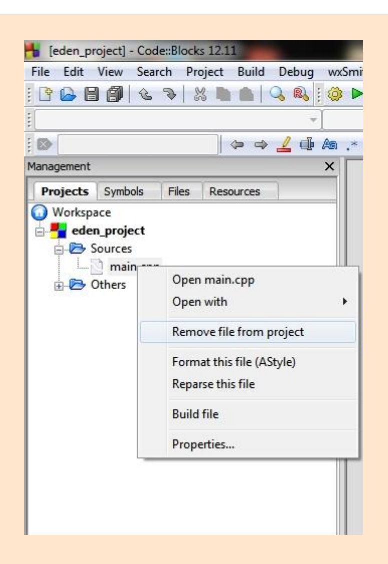
إن لم يشتغل البرنامج و لازال المترجم يشير دائما إلى عدم وجود الملف `SDL.dll`، جرب نسخ هذا الأخير و لصقه في المجلدين Debug و Release من مجلّد المشروع، أي في نفس المكان الذي يتواجد به الملف التنفيذي.
أريد أن أنوّهك بأن المشروع الذي تم انشاؤه هو خاص باللغة C++ (لأنها اللغة التي يتم اختيارها تلقائيا من بيئة التطوير، كون أن هذه الأخيرة تم تطويرها للعمل بلغة الـC++)، سنقوم إذا بتحويل هذا المشروع من C++ إلى C ببساطة.
توجّه إلى ملفات المشروع، ستجد الملف `main.cpp` قم بتغيير اسمه إلى `main.c`.



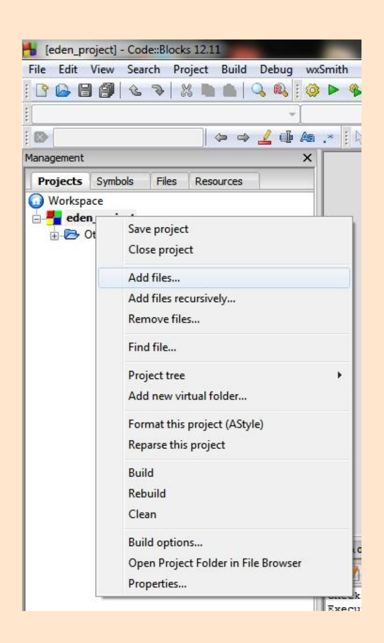
أدخل الآن إلى Code::Blocks، ستظهر لك على الأرجح النافذة التالية :



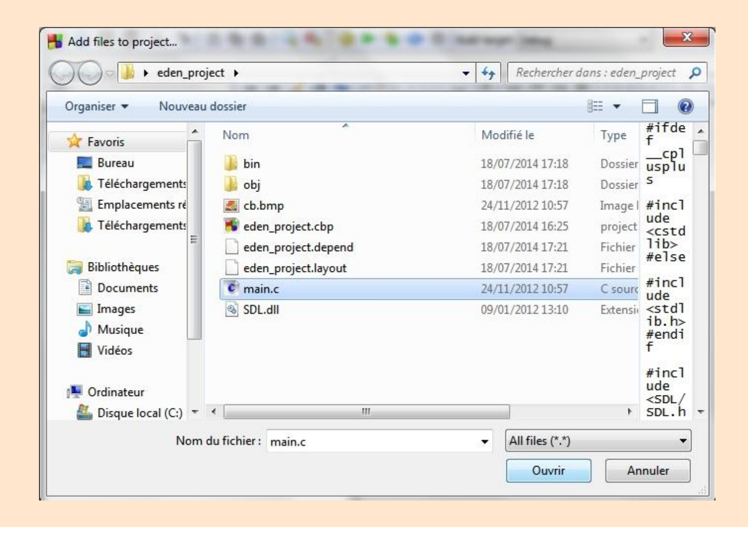
اضغط على الزر "لا" (No).
توجه إلى القائمة اليسارية، و قم بالنقر باليمين على الملف `main.cpp` و اختر حذفه من المشروع :



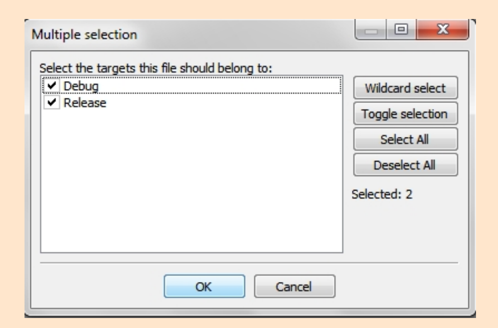
اضغط على اسم المشروع، و اطلب إضافة ملف جديد :



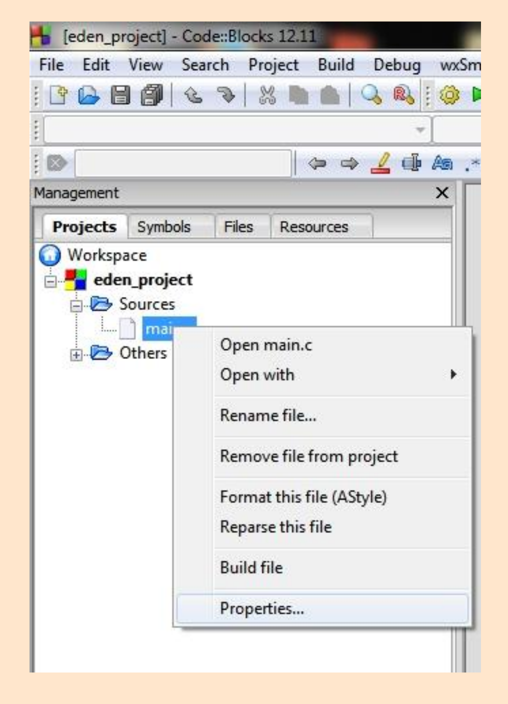
اختر الملف `main.c` من ملفات المشروع :



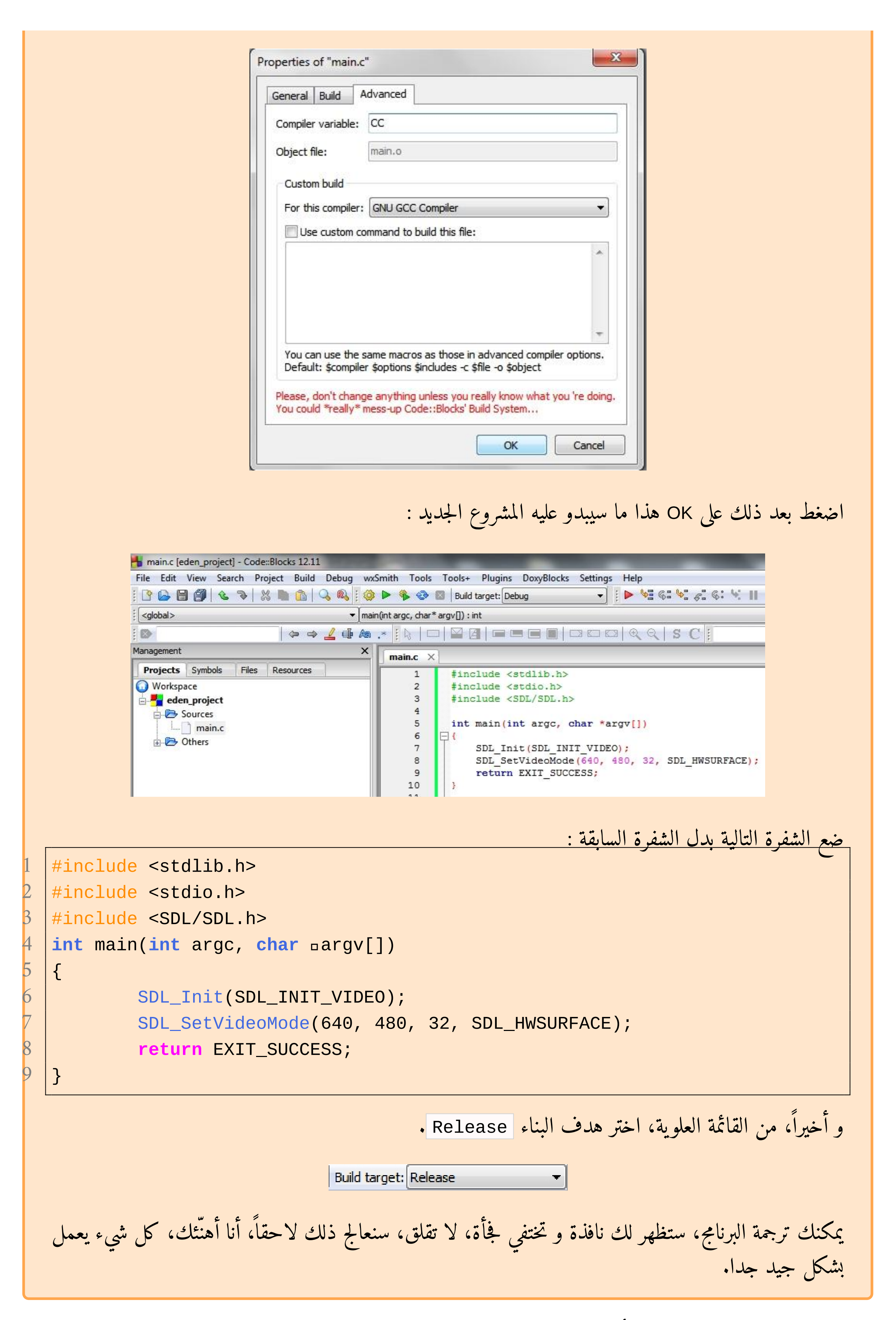
ستظهر لك نافذة أخرى، قم باختيار Next، ثم انقر على "موافق" (OK).



أنقر باليمين مجددا على الملف `main.c` و اختر "خصائص" (Properties) :



توجه إلى القائمة Advanced، ستجد Compiler variable، غيّرها من CPP إلى CC كالتالي :



اضغط بعد ذلك على OK هذا ما سيبدو عليه المشروع الجديد :

ضع الشفرة التالية بدل الشفرة السابقة :

```c
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
int main(int argc, char *argv[])
{
        SDL_Init(SDL_INIT_VIDEO);
        SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
        return EXIT_SUCCESS;
}
```

و أخيراً، من القائمة العلوية، اختر هدف البناء `Release`.

Build target: Release

يمكنك ترجمة البرنامج، ستظهر لك نافذة و تختفي فجأة، لا تقلق، سنعالج ذلك لاحقاً، أنا أهنّئك، كل شيء يعمل بشكل جيد جدا.

يمكنك مسح الملف `.bmp` لأننا لسنا بحاجة إليه. بالنسبة للملف `main.c`، يمكنك الآن استبدال محتواه بالشفرة

التالية :

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
int main(int argc, char *argv[])
{
        return 0;
}
```

إنها شفرة مبدئية، تشبه الشفرات التي تعوّدنا عليها (تضمين `stdlib.h` و `stdio.h` ثم `main`). الشيء الوحيد الذي تغيّر هو تضمين الملف `SDL.h`. إنّه ملف رأسيّ يستكلف نفسه بتضمين كل الملفات الرأسية الخاصة بالمكتبة SDL.

إنشاء مشروع SDL في ++Visual C

استخراج ملفات الـSDL

من الموقع الرسمي، قم بتنزيل آخر نسخة من المكتبة من قسم "Development Libraries" و اختر نسخة Visual C++ 2005 Service Pack 1.

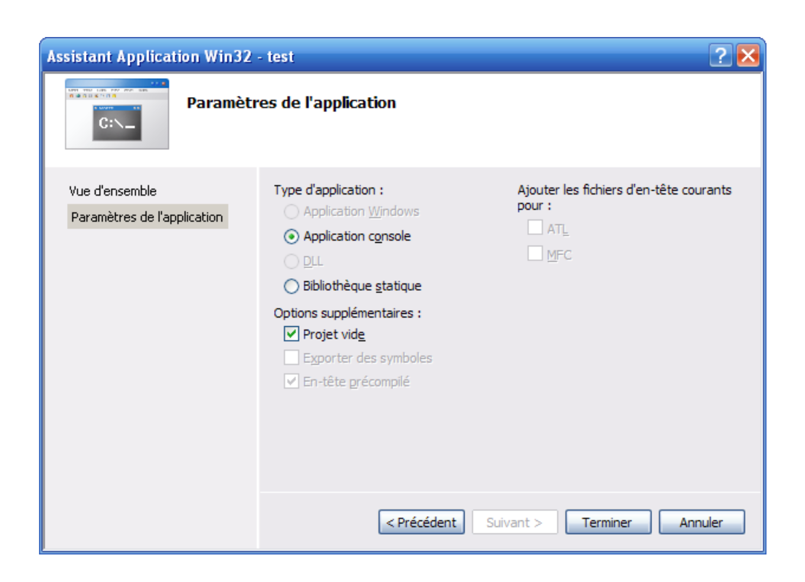
افتح الملف `.zip`.
إنه يحتوي على التوثيق (في المجلد `docs`)، الملفات `.h` (في المجلد `include`)، و الملفات `.lib` (في المجلّد `lib`) المكافئة للملفات `.a` بالنسبة لمترجم Visual. ستجد أيضاً الملف `SDL.dll` في المجلّد `lib`.

- انسخ الملف `SDL.dll` إلى مجلّد المشروع.
- انسخ الملفات `.lib` إلى المجلّد `lib` الخاص بـ++Visual C. بالنسبة لي، أنا أتكلم عن المجلّد `C:\Program Files (x86)\Microsoft Visual Studio 8\VC\lib`.
- انسخ الملفات `.h` إلى المجلّد `includes` الخاص بـ++Visual C. أنشئ مجلّداً `SDL` في المجلد `includes` لجمع الملفات `.h` الخاصة بالـSDL فيه. بالنسبة لي، سأضع تلك الملفات في المجلد : `C:\Program Files (x86)\Microsoft Visual Studio 8\VC\include\SDL`.

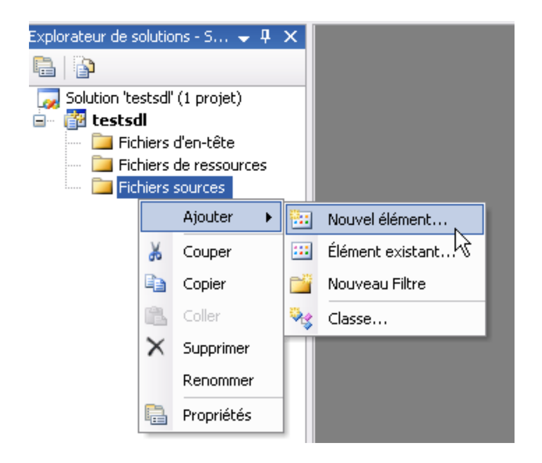
إنشاء SDL مشروع جديد

في الـ++Visual C أنشئ مشروعاً من نوع Win32 console Application. سمّه مثلاً `testsdl` ثم اضغط على "موافق".

ستفتح نافذة مساعدة. توجه إلى Application parameters و تأكد من أن الخانة `Empty project` مختارة.



لقد تمّ إنشاء المشروع إذا. إنّه فارغ. أضف إليه ملفاً جديداً و ذلك بالنقر على `Source files` ثم `Add` ثم `New element` :



حينما تفتح نافذة جديدة، أطلب إنشاء ملف جديد من نوع `C++ File (.cpp)`، قم بتسميته `main.c`. بوضعك للامتداد `.c` فإن الـVisual سيقوم بإنشاء ملف C و ليس C++.

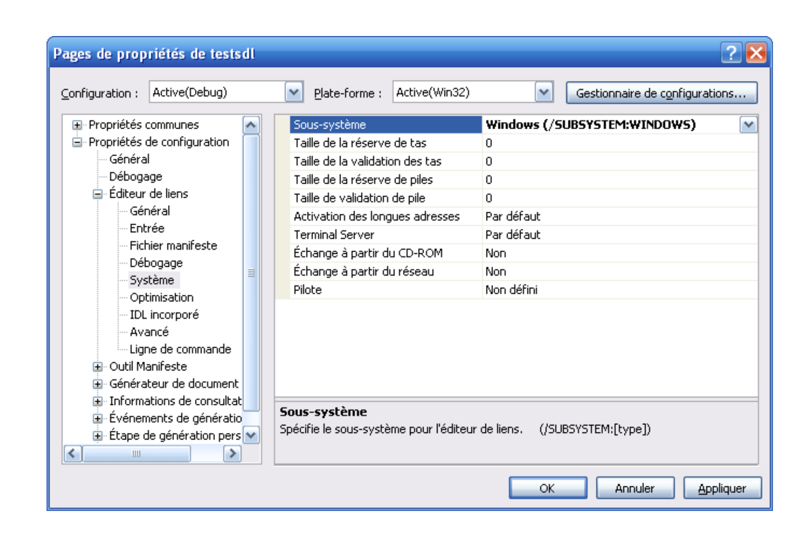
أكتب (أو انسخ/ألصِق) الشفرة المبدئية الذي وضعتها أعلاه، في الملف الجديد الذي أنشأته.

تخصيص مشروع SDL في Visual C++

التعديل على المشروع أصعب قليلاً مما هو الحال مع الـCode::Blocks، لكن بقليل من التركيز، ستتمكن من فعله. توجّه إلى خصائص المشروع من `Project` / `testsdl properties`.

- في القسم `C / C++` / `Code generation` عدّل قيمة الـ`Runtime Libraries` إلى `DLL multithread (/MD)`.
- في القسم `C / C++` / `Advanced` اختر `Compilation as` و ضع القيمة `Compile as C code (/TC)` (و إلا فإن Visual سيترجم البرنامج كأنّه ملف C++ و ليس كملف C).

- في القسم `Link editor` / `Input` عدّل قيمة الـ `Additional dependencies` لكي تضيف `SDL.lib` و `SDLmain.lib`.
- في القسم `Link editor` / `System` عدّل قيمة الـ `Sub-System` إلى الـ `Windows`.



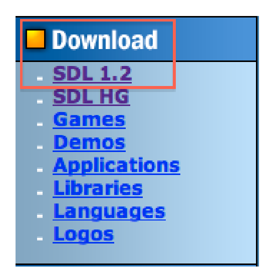
قم بالضغط على "موافق" لحفظ التغييرات.

يمكنك الآن الترجمة و ذلك بالذهاب إلى `Generate` ثم `Generate solution`.

ستجد الملف التنفيذي الذي يتواجد بمجلّد المشروع (أو بمجلد داخلي يسمى `Debug`) و لا تنس أنه على الملف `SDL.dll` أن يتواجد في نفس المجلّد الذي يتواجد به الملف التنفيذي. أنقر مرّتين على `.exe`، إذا سار كلّ شيء على ما يرام، فلن يحصل أيّ شيء، و إلّا فسيحدث خطأ إذا لم يكن الملف `SDL.dll` في نفس المجلّد.

4.20 إنشاء مشروع SDL : Mac OS (Xcode)

فلتقم بتنزيل النسخة 1.2 من الـSDL، و ذلك من خلال الجزء "Download" أسفل يسار الموقع، كالتالي :



في أسفل الصفحة ستجد قسماً يدعى "Runtime Libraries". نزّل الملف الذي يتناسب مع هندسة معالج جهازك (Intel أو PowerPC)، هذا ما ستوضحه الصورة الموالية. إن كنت تريد معرفة هندسة المعالج، يمكنك الذهاب إلى القائمة "Apple" في أعلى اليسار و النقر على "About this Mac". في السطر "Processor" ستجد إما Intel أو PowerPC.

Source Code:

SDL-1.2.15.tar.gz - GPG signed
SDL-1.2.15-1.src.rpm - GPG signed
SDL-1.2.15.zip - GPG signed

Runtime Libraries:

Linux:
SDL-1.2.15-1.i386.rpm
SDL-1.2.15-1.x86_64.rpm

Win32:
SDL-1.2.15-win32.zip
SDL-1.2.15-win32-x64.zip (64-bit Windows)

Mac OS X:
SDL-1.2.15.dmg (Intel 10.5+)
SDL-1.2.15-OSX10.4.dmg (Intel/PPC 10.4)

Development Libraries:

Linux:
SDL-devel-1.2.15-1.i386.rpm
SDL-devel-1.2.15-1.x86_64.rpm

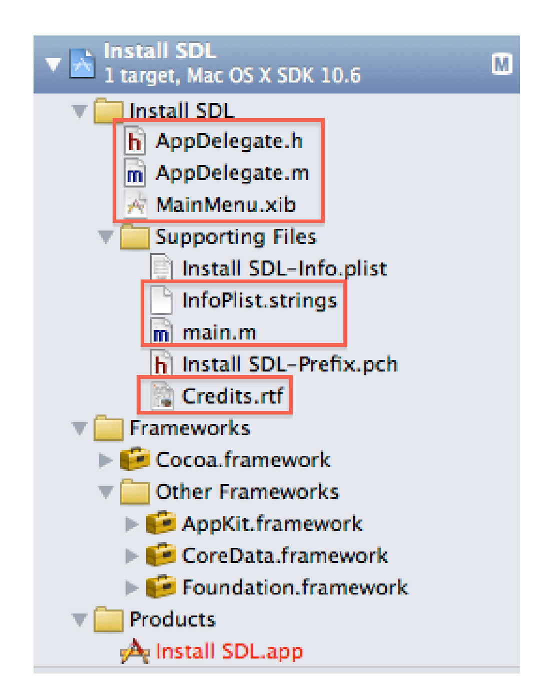
Win32:
SDL-devel-1.2.15-VC.zip (Visual C++)
SDL-devel-1.2.15-mingw32.tar.gz (Mingw32)

حينما يتم تنزيل الملف، انقر عليه مرتين، يفترض أن يفتح لوحده. ستجد بهذا المجلّد مجلّدا `SDL.framework` قم بنسخه و لصقه في المجلد `/Library/Frameworks`.

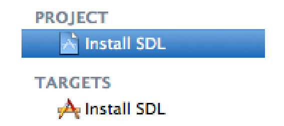
إنتهى، المكتبة مسطّبة الآن !
ستجد مجلّدا آخر اسمه `devel-lite` أتركه مفتوحا، سنعود إليه لاحقاً.

الآن قم بانشاء مشروع جديد "Cocoa Application"، اضغط على "Next". في `Product Name` قم بتسمية المشروع (كـ"SDL" مثلاً). و في `Company Identifier`، ضع ما تريد (كاسم مستعار لك مثلا). أترك الباقي كما هو ثم اضغط على "Next". اختر أين تريد وضع المشروع. سيتم إنشاء مجلّد بطريقة تلقائية و ليس عليك إنشاء واحد بنفسك ووضع ملفات المشروع بداخله.

ما إن يتم إنشاء المشروع، قم بالتخلص من الملفات التي لا تحتاجها : `AppDelegate.h`، `AppDelegate.m`، `MainMenu.xib`، `InfoPlist.strings`، `main.m` و `Credits.rtf` :

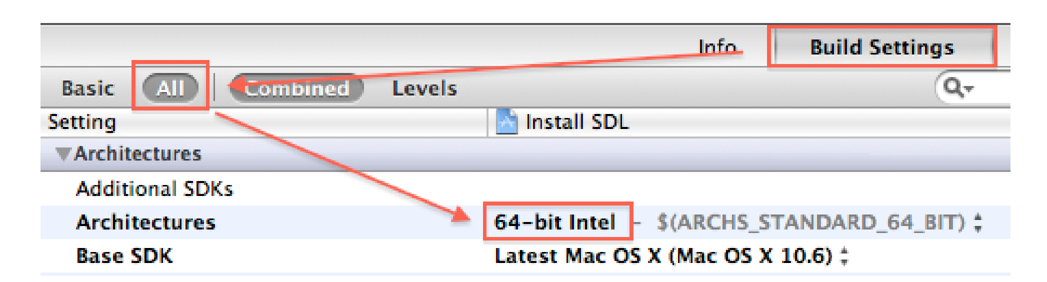


اختر المشروع من التفرع الشجري اليساري (القسم `Install SDL` من الصورة الموالية) في الشجرة الثانية اختر اسم مشروعك من قسم `PROJECT` و ليس من `TARGETS` :

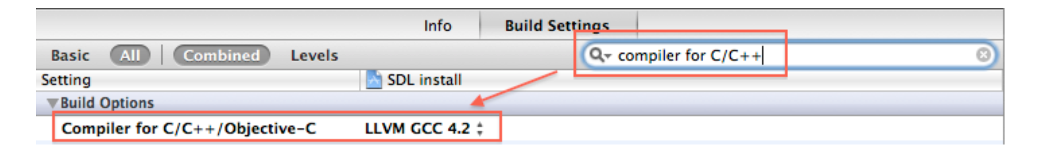


يمكنك أيضاً تغيير الـlocalisation من `English` إلى `French`. اختر `English`، أنقر على `-` للمسح و على `+` لإضافة `French`، هذا يعود إليك و لست مضطراً للقيام بذلك.

سنقوم الآن بتخصيص المشروع على نظام 32 bits (لأن المكتبة لا تشتغل على أنظمة 64 bits)، و سنقوم بإضافة المسارات من أجل الـframeworks، و للملفات الرأسية أيضاً. اضغط على `Build Settings` ثم `All` ثم في `Architectures` انقر على `64-bit Intel` و اختر `32-bit Intel` :



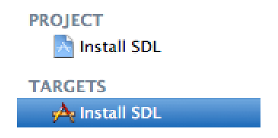
ما إن تفعل ذلك، اختر `LLVM GCC 4.2` من السطر `Compiler for C/C++/Objective-C`.



اذهب إلى منطقة البحث في أعلى اليمين، و اكتب ”search paths“، يجدر بك أن تجد سطرين مهمّين بالنسبة لنا و هما `Header search paths` و `Framework search paths` . انقر مرتين على الجهة اليمنى للسطر `Framework search paths` أنقر على علامة `+` و أضف المسار `/Library/Frameworks` . بالنسبة للسطر `Header Search paths` أضف المسار `/Library/Frameworks/SDL.framework/Headers` .

Framework Search Paths	**/Library/Frameworks**
Header Search Paths	**/Library/Frameworks/SDL.framework/Headers**

اختر الآن ”هدفك“، و هذه المرة من قسم `TARGETS` :

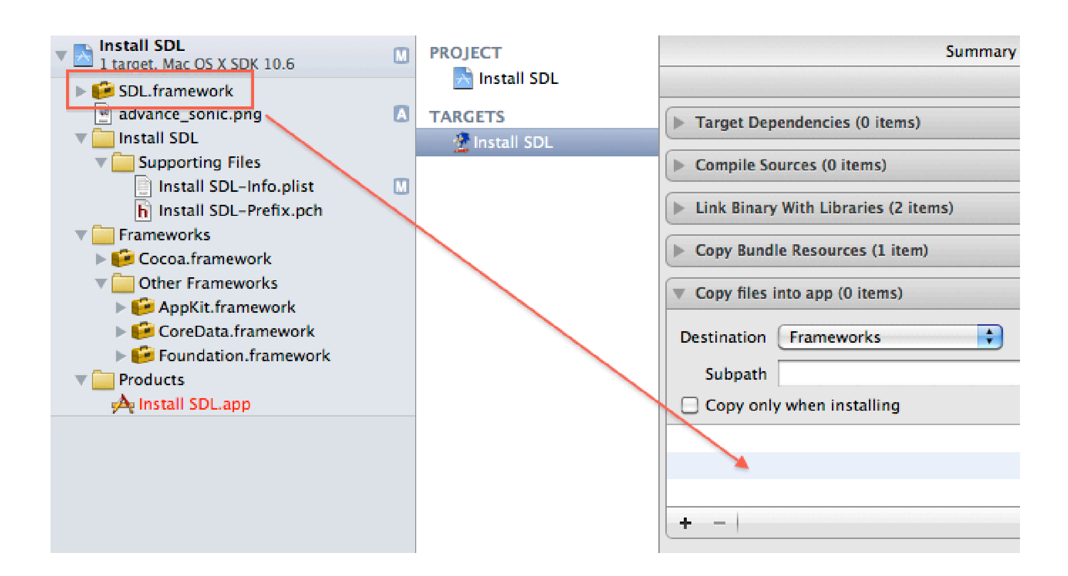


توجه إلى `Summary` ، في المنطقة `Application Category` يمكنك وضع ما تشاء، لن يغير هذا شيئا كبيراً، لأنه ينفع من أجل الـAppStore فقط. عدّل السطر `Main Interface` و ضع ”SDLMain“. بالنسبة للـ`App Icon` فاسمه يدل عليه، فهو يسمح لك بتحديد أيقونة لبرنامجك. يكفي سحب ثم تحرير الصورة المُراد استعمالها كأيقونة. بالنسبة للمنطقة `Linked Frameworks and Libraries` سنقوم بإضافة الـframework الخاص بنا `SDL.framework` ، أنقر على `+` في منطقة البحث ، أكتب ”SDL“، حين تجده في القائمة، أنقر على `Add` ، إن لم تجده فهذا يعني أنك لم تقم بوضعه في المجلّد المناسب (`/Library/Frameworks`).

الأيقونات في Mac OS هي بصيغة `.icns` . إن استعملت صيغة أخرى فستلاحظ أن الأيقونة لا تظهر. إن أردت تحويل صيغة صورة عادية إلى أيقونة استعمل برنامج **Icon Composer**. المتواجد في المجلد `/Developer/Applications/Utilities` ، يكفي أن تسحب الأيقونة إلى المربع المخصص لها ثم تحفظها.

في المنطقة `Info` يمكننا أن نشير إلى العديد من المعلومات في البرنامج، يمكنك الإطلاع عليها أكثر من خلال قراءة الملفات التوثيقية الخاصة بـApple. الشيء الوحيد الذي يمكنك التعديل عليه هو `Localization` من `en` إلى `fr` ، كما يمكنك تعديل الـ`Copyright` و وضع ما تريد.

توجّه الآن إلى `Build Phases` و انقر على `Add Build Phase` / `Copy Files` في أسفل يمين النافذة، اضغط على `Copy Files` و غيره إلى `Copy frameworks into app` . في `Destination` اختر `Frameworks` لتضيف الخاصة بك، قم بسحبها من التفرع الشجري اليساري و افلاتها في المنطقة `Build phase` ، كما يظهر بالصورة :



أنصحك بترتيب كل الـframeworks الخاص بك في مجلد `Framework` و هذا لكي يسهل عليك إيجادها.
و أيضاً بالنسبة للشفرات المصدرية، أنصحك بترتيبها في مجلدات ليسهل الوصول إليها. لإنشاء مجلد انقر باليمين على الشجرة اليسارية و اختر `New Group` ثم اسحب الملفات إلى داخلها.

سنقوم الآن بإضافة الملفات `SDLMain.h` و `SDLMain.m`، توجه إلى المجلد `devel-lite` المفتوح مسبقا و قم بإضافة الملفين إلى المشروع. إذا ظهرت لك نافذة تحديد خصائص النسخ، قم باختيار `Copy items into destination group’s folder (if needed)`.

آخر شيء : أنشئ ملفا `main.c`. توجه إلى القائمة `File` / `New` / `New File` ثم إلى `C and C++`، اختار `C File` ثم "Next". قم بتسمية الملف و هاقد أكملت.

5.20 إنشاء مشروع SDL : GNU/Linux

لن يستعملون بيئة تطويرية من أجل الترجمة، فعليهم بتغيير خواص المشروع (فالعملية مشابهة لما كنت قد شرحت).
بالنسبة لمن يستعمل Code::Blocks، فالطريقة هي نفسها التي شرحتها سابقاً.

> ماذا عن الذين يقومون بترجمة الشفرات يدويا ؟

قد يوجد بين القراء من اعتاد على ترجمة الشفرات يدويا بالاستعانة بـMakefile (ملف يساعد على عملية الترجمة).
إذا كانت هذه حالتك، فأدعوك لتحميل Makefile الذّي يُمكن أن يُستخدم لترجمة مشاريع الـSDL.

http://www.siteduzero.com/uploads/fr/ftp/mateo21/makefile_sdl

الشيء الوحيد المختلف، هو إضافة المكتبة SDL إلى محرّر الروابط (`LDFLAGS`). يجدر بك أن تكون قد نزّلت المكتبة و ثبّتها في مجلّد ملفات المترجم، بنفس طريقة Windows (المجلدا `include/SDL` و `lib`).

بعد ذلك يجب عليك أن تكتب الأوامر التالية في الكونسول :

```
make            # To compile the project
make clean      # To delete compilation files (useless .o files)
make mrproper   # To delete all files except source ones
```

ملخّص

- الـSDL مكتبة منخفضة المستوى، تسمح بإنشاء نوافذ و التعامل مع الرسوميات 2D.
- المكتبة ليست مسطّبة تلقائيا في الحاسوب، يجب عليك أن تنزّلها بنفسك و تقوم بتخصيص البيئة التطويرية لتعمل معها.
- المكتبة حرة و مجّانية، مما يسمح باستعمالها السريع و الدائم.
- توجد آلاف المكتبات الأخرى، و كثير منها ذو جودة عالية جداً. و قد تم اختيار المكتبة SDL لبقيّة هذا الكتاب لأجل سهولتها. لمن يريد بناء واجهات رسومية بنوافذ، أزرار و قوائم، فأنا أنصحه بالمكتبة +GTK مثلا.

الفصل 21

إنشاء نافذة و مساحات

في الفصل السابق، قمنا بالإلمام حول أهم المميزات التي تمنحها المكتبة SDL. يجدر بك أن تكون قد ثبتّ المكتبة، و تعلّمت كيفية إنشاء مشروع جديد يشتغل بشكل جيد. على الرغم من أنّه كان فارغا.

سندخل في مضمون موضوعنا في هذا الفصل. سنقوم بتطبيق أساسيات لغة الـC مع SDL. كيف يتم تحميل الـSDL ؟ كيف يتم فتح نافذة بالأبعاد التي نريد ؟ كيف نرسم داخل النافذة ؟

لدينا أمور كثيرة لنعرفها، فهيّا بنا !

1.21 تحميل و إيقاف الـSDL

العديد من المكتبات المكتوبة بلغة الـC، تستلزم أن يتم تحميلها ثم غلقها حين ننتهي منها، و ذلك لاستعمال دوال محددة. المكتبة SDL من بين هذه المكتبات.

بالفعل، فالمكتبة تحتاج أن يتم تحميل عدد معيّن من المعلومات إلى الذاكرة العشوائية لتستطيع أن تشتغل بشكل صحيح. يتم هذا التحميل بشكل حيّ باستعمال الدالة `malloc` (إنّها مهمّة جدّا هنا !). و كما تعلم فإن قلت `malloc` ، سأقول كذلك `free` !
يجب عليك تحرير الذاكرة التي حجزتها و لم تعد بحاجة إليها. إن لم تفعل، فالبرنامج يمكن أن يأخذ حيّزاً كبيراً من الذاكرة بدون فائدة، و يمكن لذلك أحيانا أن يدرّ بنتائج كارثية. تخيّل القيام بحلقة غير منتهية من `malloc` دون قصد، في بضع ثوان ستسدّ كلّ الذاكرة !

هاهما الدالتان الأولتان الخاصّتان بالـ`SDL` اللتان يجب عليك أن تعرفهما :

- `SDL_Init` : تحميل المكتبة في الذاكرة العشوائية (باستخدام الـ`malloc`).
- `SDL_Quit` : تحرير المكتبة من الذاكرة (باستعمال الـ`free`).

أي أن أوّل شيء يجب أن تقوم به في البرنامج هو استدعاء `SDL_Init` ، و آخر شيء هو استدعاء `SDL_Quit` .

SDL_Init : تحميل المكتبة SDL

الدالة `SDL_Init` تستقبل معاملا. إذ يجب أن يتم تحديد أي جزء من المكتبة نريد تحميله.

؟ آه حقّا ! هل الـSDL تتكون من كثير من الأجزاء ؟

نعم بالطبع ! فهناك جزء من المكتبة يتعامل مع الشاشة، و آخر يتعامل مع الصوت، إلخ.

توفّر لنا المكتبة عدداً من الثوابت التي تسمح لنا بتحديد اسم الجزء الذي نريد تحميله من المكتبة.

الثابت	الشرح
SDL_INIT_VIDEO	تحميل الجزء الخاص بالعرض (الفيديو)، إنه الجزء الذي نحمله غالباً.
SDL_INIT_AUDIO	تحميل الجزء الخاص بالصوت، هذا ما يسمح لك مثلا بتشغيل الموسيقى مثلا.
SDL_INIT_CDROM	تحميل الجزء الخاص بقارئ القرص المضغوط، و ذلك للتحكم به.
SDL_INIT_JOYSTICK	تحميل الجزء الخاص بجهاز التحكم Joystick.
SDL_INIT_EVERYTHING	تحميل كل الأجزاء التي ذكرتها سابقا.

إذا استدعيت الدالة بهذا الشكل

```
SDL_Init(SDL_INIT_VIDEO);
```

فإن نظام العرض سيتم تحميله في الذاكرة، فيمكنك أن تفتح نافذة و ترسم فيها، إلخ.
كل ما قمنا به هو إعطاء عدد إلى الدالة `SDL_Init` بالاستعانة بثابت. أنت لا تعرف أي عدد هو، و هذا أمر جيد. إذ أنك غير مجبر على حفظ العدد، بل التعبير عنه باسم الثابت فقط.

الدالة `SDL_Init` تقرأ العدد و هكذا تحدد الأنظمة الواجب تحميلها.

الآن لو تكتب :

```
SDL_Init(SDL_INIT_EVERYTHING);
```

ستقوم بتحميل كل أنظمة الـSDL، لا تقم بهذا إلا في حالة كنت بالفعل تحتاج إلى كلّ شيء، ليس جيداً إثقال الحاسوب بوحدات لا فائدة منها.

؟ ماذا لو أردت تحميل الصوت و الفيديو فقط. هل يجدر بي استخدام `SDL_INIT_EVERYTHING` ؟

لن تستعمل `SDL_INIT_EVERYTHING` ، من أجل تحميل وحدتين، هذا جنون ! لحسن الحظ، يمكننا تجميع الخيارات بواسطة الرمز `|` .

```
// Loading the video and the audio
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO);
```

كما يمكنك وضع ثلاثة دون مشاكل :

```
// Loading the video, the audio and the timer
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER);
```

هذه "الخيارات" التي نبعثها للدالة `SDL_Init` نسميها بـالأعلام (Flags). هذه الكلمة نستعملها كثيراً في علوم الحاسوب.
تذكّر إذا أن الإشارة `|` خاصة بدمج الخيارات. إنها تشبه الإضافة إلى حدّ ما.

SDL_Quit : إيقاف المكتبة SDL

هذه الدالة سهلة الاستعمال لأنها لا تحتاج إلى أي معامل :

```
SDL_Quit();
```

كل الأنظمة سيتم إيقافها و يتم تحرير الذاكرة.
باختصار، هذه الدالة أداة للخروج من المكتبة بشكل نظيف، و لنقل للخروج من برنامجك.

نموذج عن برنامج SDL

باختصار، هذا ما يبدو عليه برنامج SDL في نسخته الأبسط :

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
int main(int argc, char *argv[])
{
        SDL_Init(SDL_INIT_VIDEO); // Starting the SDL (Here we load the video
            system)
        SDL_Quit(); // Stopping the SDL (Freeing the memory).
        return 0;
}
```

هذا نموذج عن برنامج بسيط، عبارة عن مخطط لبرامج SDL التي نكتبها. في الواقع، البرنامج الحقيقي يكون ممتلأً كثيرا إذ يحتوي عدّة استدعاءات لدوال، تقوم بدورها بمزيد من الاستدعاءات.
الأمر المهمّ في النهاية، هو أنّ SDL يجب أن تُحمّل في البداية و تُغلق عندما لا تصبح بحاجة إليها.

معالجة الأخطاء

الدالة `SDL_Init` تقوم بإرجاع قيمة :

- -1 : في حال وجود خطأ.
- 0 : في حالة عدم وجود أي خطأ.

لست مجبراً، لكن يمكنك اختبار القيمة المُرجعة. قد تكون طريقة جيّدة لمعالجة الأخطاء في برنامجك، و هذا ما سيساعدك على حلّها.

؟

لكن كيف أقوم بإظهار الخطأ الحادث ؟

سؤال وجيه ! ليس لدينا كونسول الآن، كيف نخزّن و نعرض رسائل الخطأ ؟

هناك حلّان :

- يمكننا التعديل على خاصيات المشروع، لكي نسمح له باستعمال الكونسول أيضاً. ستتمكّن في هذه الحالة من استخدام الدالة `printf`)،
- أو نكتب الأخطاء في ملف. تستخدم الدالة `fprintf`.

لقد اخترت أن نكتب في ملف. و بهذا فإن العمل على ملف يحتاج إلى فتح هذا الأخير بـ`fopen` و غلقه بـ`fclose`، و الأمر أقلّ سهولة من استعمال الـ`printf`.
لحسن الحظ، هناك طريقة أسهل و هي استعمال مخرج الأخطاء القياسي.

يوجد متغير `stderr` معرّف في `stdio.h` يقوم بالتأشير نحو المنطقة التي يُمكن أن يُكتب فيها الخطأ. غالبا في الويندوز، هذه المنطقة عبارة عن ملف يحمل الاسم `stderr.txt`. بينما في الليناكس فإن الأخطاء غالباً ما يتم إظهارها على الكونسول. هذا المتغير يتم إنشاؤه تلقائيّا في بداية البرنامج و يتم حذفه في نهايته، أي أنك لست مجبراً على استعمال `fopen` و `fclose`.
يمكنك استعمال الدالة `fprintf` على `stderr` بدون استعمال `fopen` و `fclose` :

```c
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
int main(int argc, char *argv[])
{
        if (SDL_Init(SDL_INIT_VIDEO) == -1) // Starting the SDL, if there's an error :
        {
                fprintf(stderr, "Error while initializing SDL : %s\n", SDL_GetError()); // Writing the error
```

```
9              exit(EXIT_FAILURE); // We exit the program
10         }
11         SDL_Quit();
12         return EXIT_SUCCESS;
13 }
```

ما الجديد في هذه الشفرة المصدرية ؟

- لقد كتبنا الخطأ الذي وجدناه في `stderr` . الرمز `%s` يسمح للـSDL بالإشارة إلى تفاصيل الخطأ : الدالة `SDL_GetError` في الحقيقة تقوم بإرجاع آخر خطأ SDL.
- نخرج باستعمال الـ `exit()` . لحدّ الآن لا يوجد شيء جديد مقارنة بما جرت العادة، ستلاحظ أنني استعمل الثابت `EXIT_FAILURE` كقيمة يقوم البرنامج الرئيسي بإرجاعها، بينما استعملت في النهاية الثابت `EXIT_SUCCESS` في مكان الـ0.

ما الذي قمت به ؟ لقد قمت بتحسين الطريقة التي تعودنا أن نكتب بها الشفرة. لقد استخدمت اسم الثابت الذّي يعني "خطأ" و الذي هو نفسه بالنسبة لجميع أنظمة التشغيل. بينما الأعداد تختلف من نظام إلى آخر.
لهذا فإن الملف `stdlib.h` تسمح باستعمال ثابتتين (معرّفي `#define`) :

- `EXIT_FAILURE` : قيمة يتم إرجاعها في حالة وجود خطأ ما في البرنامج.
- `EXIT_SUCCESS` : قيمة يتم إرجاعها في حالة عدم وجود أي خطأ.

باستعمال أسماء الثوابت بدلاً من قيمها، ستضمن بأنك قد بعثت القيمة الصحيحة.
لماذا ؟ لأن الملف `stdlib.h` يتغيّر حسب نظام التشغيل الذّي أنت عليه، لذا فقيم الثوابت ستتأقلم مع النظام من دون أنّ نحتاج إلى تغيير شيء ! و هذا ما يجعل لغة الـC متوافقة مع كلّ أنظمة التشغيل (بافتراض أنّك تبرمج بالطريقة الصحيحة باستخدام الأدوات المتوفّرة، كما فعلنا هنا).

> استعمال أسماء الثوابت لا يعود علينا بكثير من النفع الآن، لكن من الأحسن استعمالها. سنقوم بذلك انطلاقاً من الآن.

2.21 فتح نافذة

حسناً، لقد تم فتح و غلق المكتبة بنجاح. الخطوة التالية التي يريد تعلّمها الجميع هي كيفية فتح نافذة !

لكي نبدأ، تأكد من أن لديك `main` تشبه هذه :

```
1 int main(int argc, char *argv[])
2 {
3         if (SDL_Init(SDL_INIT_VIDEO) == -1)
4         {
```

```
            fprintf(stderr, "Error while initializing SDL");
            exit(EXIT_FAILURE);
        }
        SDL_Quit();
        return EXIT_SUCCESS;
}
```

هذه هي حالتك لو اتبعت جيداً من بداية الفصل. حاليّا، كلّ ما سنقوم بتحميله هو نظام العرض (`SDL_INIT_VIDEO`)، هذا ما يهمّنا.

اختيار وضع العرض

أول شيء نقوم به بعد `SDL_Init`، هو تحديد وضع العرض الذي نريد استعماله، أي الدقة (Resolution)، عدد الألوان بالإضافة إلى خصائص أخرى.

من أجل هذا سنستعمل الدالة `SDL_SetVideoMode` التي تستقبل 4 معاملات :

- عرض النافذة التي نريدها (pixels)،
- طول النافذة التي نريدها (pixels)،
- عدد الألوان القابلة للعرض (bits/pixel)،
- الخيارات (الأعلام).

لا أعتقد أن طول و عرض النافذة يحتاجان إلى شرح، بينما عدد الألوان و الأعلام هما المعاملان الأكثر أهميّة.

- **عدد الألوان** : هو العدد الأقصى للألوان التي يمكن أن تظهر في النافذة. إن كنت من عشاق ألعاب الفيديو، ستكون معتاداً على هذا الأمر. فإن قيمة 32 bits/pixel تسمح بإظهار ملايير الألوان، بينما إنه من الممكن أن نختار قيمة أقل كـ16 bits/pixel (تسمح بعرض 65536 لون) أو حتى 8 bits/pixel (تسمح بعرض 256 لون مختلف)، هذا الأمر مفيد حينما تريد برمجة تطبيقات من أجل جهاز بسيط كالـPDA أو الهاتف المحمول.
- الخيارات : تماما مثل الـ`SDL_Init`، علينا باستعمال أعلام من أجل تعريف خصائص. هذه أهم الأعلام التي يمكنك استعمالها (يمكنك استعمال العديد منها، يتم التفريق بينها باستعمال الرمز `|`) :
  - `SDL_HWSURFACE` : المعطيات سيتم حفظها في الذاكرة الرسومية للبطاقة 3D. الشيء الجيد : إنها الذاكرة الأكثر سرعة. الشيء السيّء : يصعب إيجاد مساحة شاغرة في هذا النوع من الذاكرة مقارنة بالأخرى (`SDL_SWSURFACE`).
  - `SDL_SWSURFACE` : المعطيات يتم حفظها في ذاكرة النظام (أي في الـRAM)، الشيء الجيد : يوجد الكثير من المكان في هذه الذاكرة. الشيء السيء : أقل سرعة و أقل كفاءة.
  - `SDL_RESIZABLE` : ستصبح مقاييس أبعاد النافذة قابلة للتعديل، لأنها ليست كذلك تلقائيا.
  - `SDL_NOFRAME` : لن يصبح للنافذة أية حواش أو شريط علوي لكتابة عنوان النافذة.

– `SDL_FULLSCREEN` : نمط الشاشة الكاملة. لن تستطيع رؤية أية نافذة أخرى لأن نافذة البرنامج الحالي تهيمن على كلّ الشاشة، مع تعديل دقّة الشاشة في حالة الضرورة.
– `SDL_DOUBLEBUF` : وضع double buffering، تقنية مستعملة بكثرة في برمجة الألعاب ثنائية الأبعاد. تقضي بأن يكون تحرّك الأشياء على الشاشة مرناً، لأنه إن لم يكن كذلك، سيكون التحرّك سيئاً، سأشرح هذا الأمر بالتفصيل لاحقاً.

إذا كتبت الشفرة المصدرية التالية :

```c
SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
```

فإنه سيقوم بفتح نافذة ذات أبعاد 640×480، و بعدد ألوان 32 bits/pixel (ملايير الألوان)، و ستم تحميل النافذة على الذاكرة الرسوميّة (إنّها الأسرع، لذلك نفضّل استعمالها).

كمثال آخر لو نأخذ :

```c
SDL_SetVideoMode(400, 300, 32, SDL_HWSURFACE | SDL_RESIZABLE | SDL_DOUBLEBUF);
```

هذه الشفرة تفتح نافذة مقاييس أبعادها قابلة للتعديل، بأبعاد ابتدائية 400×300، و بعدد ألوان 32 bits/pixel كما أن تقنية double buffering مفعّلة.

هذه أوّل شفرة مصدرية بسيطة يمكنك تجريبها :

```c
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
int main(int argc, char *argv[])
{
        SDL_Init(SDL_INIT_VIDEO);
        SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
        SDL_Quit();
        return EXIT_SUCCESS;
}
```

لقد اخترت أن أسحب معالجة الأخطاء لتبسيط الشفرة، لكن بالنسبة لك، يجب عليك أن تكتب برامج كاملة و أخذ كلّ الاحتياطات اللازمة لمعالجة الأخطاء.

قم بتجريب الشفرة. ما الذي يحصل ؟ تظهر النافذة و تختفي بسرعة البرق.
الحقيقة أن استدعاء الدالة `SDL_SetVideoMode` يليه مباشرة استدعاء الدالة `SDL_Quit` التي تقوم بإنهاء كلّ شيء.

توقيف البرنامج للحظات

> ؟ ما العمل كي تقوم النافذة بالانتظار و لا تختفي مباشرة ؟

يجب أن نفعل ما تفعله جميع البرامج، سواء كانت ألعابا أو غير ذلك، حلقة تكرارية غير منتهية. في الواقع، بمساعدة حلقة غير منتهية بسيطة سنمنع البرنامج من التوقف. لكن هذه الطريقة فعالة جدّا لدرجة أنّه لا يمكننا إيقاف البرنامج (إلا إذا أجبرناه باستدعاء المعالج، لكنّها تبقى طريقة عنيفة لإنهاء عمل برنامج).

هذه شفرة تعمل، لكنّي أدعوك إلى عدم تجريبها، أعطيها لك فقط كشرح :

```
int main(int argc, char *argv[])
{
        SDL_Init(SDL_INIT_VIDEO);
        SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
        while(1);
        SDL_Quit();
        return EXIT_SUCCESS;
}
```

أنت تعرف الـ `while(1);` : إنّها الحلقة التكرارية غير المنتهية. بما أن 1 يساوي القيمة المنطقية "صحيح" (تذكّر المتغيرات المنطقية)، فإن الشرط صحيح دائما و بالتالي فستدور الحلقة إلى الأبد مع عدم وجود وسيلة لإيقافها. هذا ليس حلاّ جيّدا.

لكي تتمكن من توقيف النافذة كي لا تختفي فجأة بدون اللجوء إلى حلقة غير منتهية، سنستعمل دالتي التّي أنشأتها و سميتها `pause` :

```
void pause()
{
        int cont = 1;
        SDL_Event event;
        while (cont)
        {
                SDL_WaitEvent(&event);
                switch(event.type)
                {
                        case SDL_QUIT:
                        cont = 0;
                }
        }
}
```

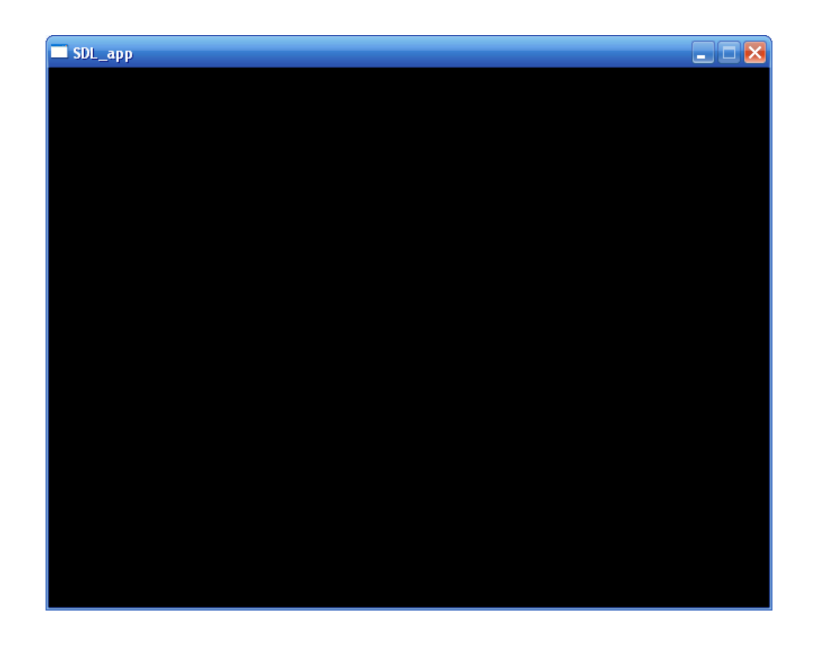
لن أشرح لك تفاصيل الدالة الآن. فهذه الدالّة تحتاج إلى ما نسميه بمعالجة الأحداث التّي سأشرحها في الفصل القادم. فإن شرحت لك كلّ شيء الآن فقد تختلط عليك الأمور ! ثِق في دالّتي الخاصّة بالتوقيف، ستتعرف على طريقة عملها قريباً.

هذا مثال عن شفرة مصدرية كاملة، يمكنك (أخيراً) تجريبها :

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
void pause();
int main(int argc, char *argv[])
{
        SDL_Init(SDL_INIT_VIDEO); // We initialize the SDL
        SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
        pause(); // We pause the program
        SDL_Quit(); // We stop the SDL
        return EXIT_SUCCESS; // We close the program
}
void pause()
{
        int cont = 1;
        SDL_Event event;
        while (cont)
        {
                SDL_WaitEvent(&event);
                switch(event.type)
                {
                        case SDL_QUIT:
                        cont = 0;
                }
        }
}
```

ستلاحظ أنني وضعت نموذج الدالة `pause` في أعلى البرنامج كي لا أضطرّ لعرض أكثر من ملف.
لقد قمت باستدعاء الدالة `pause` و هي تقوم بالدخول في حلقة تكرارية غير منتهية أذكى من السابقة. هذه الحلقة تنتهي حينما تنقر على الزر الأحمر `X` أعلى النافذة !

الصورة التالية هي عبارة عن النافذة التي نحصل عليها حين نترجم الشفرة المصدرية السابقة (النافذة ذات أبعاد 640×480).



ها قد وصلنا !

إن أردت، قم بوضع العَلم الذي يسمح بتعديل مقاييس النافذة. للعِلْم، في ألعاب الفيديو نفضّل النوافذ ذات الأبعاد الثابتة (لأنه يسهل التعامل معها)، إذا فلنترك النافذة ثابتة كما هي الآن.

> !
> احذر من العَلَم `SDL_FULLSCREEN` الخاص بوضع الشاشة الكاملة، و من العَلم `SDL_NOFRAME` الذي يقوم بإخفاء حواشي النافذة. بما أنّه لن يكون هناك شريط للعنوان، فلن نكون قادرين على الخروج من البرنامج، إلا بالاستعانة بالمعالج !
> تريّث قليلاً حتى نتعلّم معالجة الأحداث (في الفصول القادمة) و ستتمكن بعدها من الخروج من النافذة بطريقة أقل عنفاً من استدعاء المعالج.

تغيير عنوان النافذة

لحدّ الآن، النافذة أخذت عنوانا تلقائيا (و هو `SDL_app` في الصورة السابقة).
هل تريد تغييره ؟

إن الأمر بسيط للغاية، يكفي استعمال الدالة `SDL_WM_SetCaption`.
هذه الدالة تأخذ معاملين : المعامل الأول هو العنوان الذي تريد إعطاءه للنافذة، و المعامل الثاني هو العنوان الذي تريد إعطاءه للأيقونة.

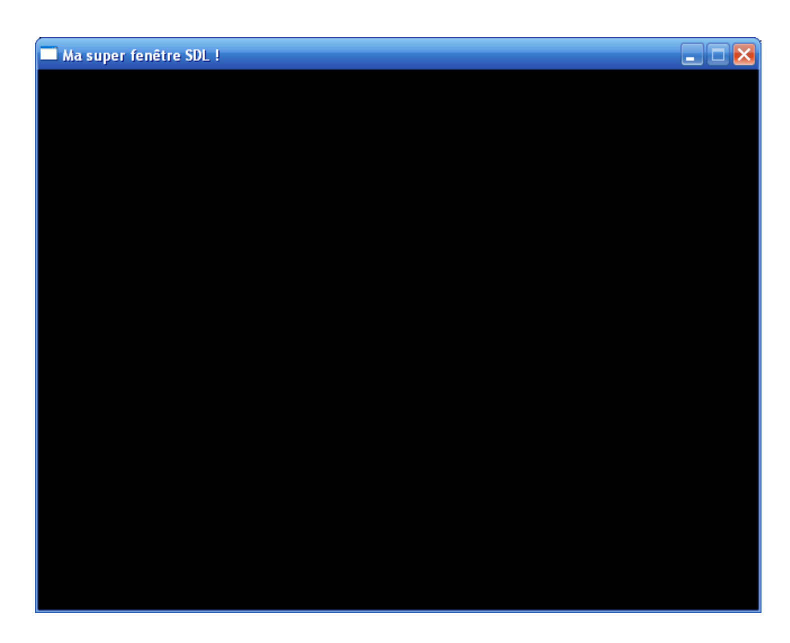
خلافاً لما يعتقده الجميع، تغيير اسم الأيقونة لا يعني تغيير صورة الأيقونة التي تظهر أعلى يسار النافذة. هذا لا يعمل دائما (حسب معرفتي، قد يعطي نتائج على الـGNU/Linux في بيئة الـGnome). شخصياً، أنا أبعث القيمة `NULL` إلى الدالة. على أية حال، يمكننا تغيير شكل الأيقونة التي تظهر أعلى يسار النافذة، لكننا سنتعلّم ذلك في الفصل القادم، لأنّ هذا الأمر ليس بمستواك بعد.

هذه نفس الـ`main` السابقة، مع إضافة الدالة `SDL_WM_SetCaption` :

```c
int main(int argc, char *argv[])
{
        SDL_Init(SDL_INIT_VIDEO);
        SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
        SDL_WM_SetCaption("Ma super fenêtre SDL !", NULL);
        pause();
        SDL_Quit();
        return EXIT_SUCCESS;
}
```

لاحظ بأنني استعملت القيمة `NULL` للمعاملات غير المهمّة بشكل كبير. بالنسبة لـC، يجب أن يتم إعطاء قيم لكل المعاملات التي تستقبلها الدوال، حتى لو كانت هذه المعاملات غير مهمة لك، فأعطها `NULL` كما فعلت أنا هنا. يبنما الـ++C تسمح بألا نعطي أساساً قيمة لبعض المعاملات الاختياريّة عندما نستدعي الدوال.

للنافذة الآن عنوان.



3.21 التعامل مع المساحات

لحد الآن تمكّنا من فتح نافذة ذات خلفيّة سوداء. ما نريد الآن هو أن نملأها ببعض الأشياء، أي أن "نرسم" فيها.

كما قلت لك في الفصل السابق، فإن المكتبة SDL هي مكتبة منخفضة المستوى، أي أنها لا توفّر لنا سوى دوال قاعدية، بسيطة جداً.
الصراحة هي أن الشكل الوحيد الذي تسمح لنا الـSDL برسمه هو المستطيل ! كلّ ما سنقوم به هو جمع بعض المستطيلات في نافذة. نسمّي هذه المستطيلات بـ**المساحات** (Surfaces)، المساحة هي الوحدة الرسومية القاعدية في الـSDL.

إنه من الممكن أن نرسم أشياء أخرى، مثل الدوائر و المثلثات، إلخ. و لكن لكي نفعل ذلك، يجب أن نكتب بأنفسنا الدوال الّتي تمكّن من فعل ذلك، برسم تلك الأشكال بيكسلا ببيكسل، و إما أن نستعمل مكتبة أخرى إلى جانب الـSDL. الأمر معقّد نوعاً ما، لكن لا تقلق، ستجد بأننا لسنا بحاجة إلى كلّ هذا في التطبيق.

مساحتك الأولى : الشاشة

في كل برامج الـSDL، توجد على الأقل مساحة عمل واحدة و هي ما نسميه بالشاشة (Screen)، و هي مساحة توافق كل النافذة، أي كلّ المساحة السوداء التي تظهر بالنافذة.

في الشفرة المصدرية، كل مساحة يتم تخزينها في متغير من نوع `SDL_Surface` . نعم، إنه نوع بيانات تم إنشاؤه من طرف الـSDL (هذا المتغير عبارة عن هيكل).

بما أن أول مساحة ننشئها هي الشاشة، فهيا بنا :

```
1 SDL_Surface *screen = NULL;
```

تلاحظ أنني قمت بإنشاء مؤشّر. لماذا أفعل هذا ؟ لأن الـSDL هي من ستقوم بحجز مكان في الذاكرة من أجل مساحتنا. المساحة بالفعل ليس لها بالضرورة دائما نفس الحجم و لهذا فعلى الـSDL أن تقوم بحجز حيّ من أجلنا (هنا، هذا يعتمد على حجم النافذة التي فتحناها).

لم أقل لك هذا من قبل، لكن الدالة `SDL_SetVideoMode` تقوم بإرجاع قيمة ! ستقوم بإرجاع مؤشّر نحو المكان بالذاكرة المخصص لمساحة الشاشة.
ممتاز، يمكننا إذا استرجاع المؤشّر في المتغير `screen` :

```
1 screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
```

المؤشّر الآن يمكن أن يساوي إحدى القيمتين :

- `NULL` : المتغير `screen` سيساوي `NULL` إذا فشلت الدالة `SDL_SetVideoMode` في تحميل أسلوب العرض الذي تم طلبه. و هذا يحصل حينما يتم اختيار دقة جد عالية أو عدد كبير جداً من الألوان، أكبر من أقصى عدد يتحمله جهازك.
- قيمة أخرى : إذا كانت القيمة مختلفة عن `NULL`، فهذا يعني أن الـSDL قامت بحجز المكان، كل شيء على ما يرام !

إنه من المستحسن هنا أن تتم معالجة الأخطاء، تماما مثلما فعلنا حينما أردنا تحميل الـSDL، هاهي إذا الدالة

الكاملة بإضافة معالجة الأخطاء للـ `SDL_SetVideoMode`.

```
int main(int argc, char *argv[])
{
        SDL_Surface *screen = NULL; // The pointer which stores the surface of
            the screen
        SDL_Init(SDL_INIT_VIDEO);
        screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE); // We try to
            open the window
        if (screen  == NULL) // If we can't, we note it and we exit.
        {
                fprintf(stderr, "Impossible to load the video mode : %s\n",
                    SDL_GetError());
                exit(EXIT_FAILURE);
        }
        SDL_WM_SetCaption("Ma super fenêtre SDL !", NULL);
        pause();
        SDL_Quit();
        return EXIT_SUCCESS;
}
```

الرسالة التي تتركها لنا الدالة `SDL_GetError`، مفيدة من أجل معرفة ما الّذي لم يعمل.

حكاية صغيرة : مرة أخطأت بينما أردت أن أفتح نافذة بأسلوب الشاشة الكاملة (Full screen)، في عوض أن أطلب الدقة 768×1024 كتبت بالخطأ 768×10244، لم أفهم لماذا لم يتم التحميل، لأنّي لم أنتبه إلى أنني كتبت 4 مرّتين (ربما كنت متعباً). و لحل المشكل ألقيتُ نظرة على الملف `stderr.txt`، توجهت إلى رسالة الخطأ و اكتشفت بأن الدقة التي طلبتها مرفوضة (شيء يثير الفضول أليس كذلك ؟).

تلوين مساحة

لا توجد 36 طريقة لملء مساحة، الحقيقة أنه توجد طريقتان :

- إما أن يتم تلوين المساحة بلون موحّد.
- إما أن يتم ملؤها عن طريق تحميل صورة.

يمكنك في الحقيقة الرسم في المساحة بيكسلا ببيكسل، لكن هذه الطريقة معقّدة، لن نراها هنا.

سنرى أولا كيف نقوم بتلوين مساحة بلون موحّد. في الفصل القادم سنتعلّم كيف نقوم بتحميل صورة.

الدالة التي تسمح بتلوين النافذة بلون موحّد هي `SDL_FillRect` (العبارة `FillRect` تعني ملء مستطيل بالإنجليزيّة). هذه الدالة تستقبل 3 معاملات و هي :

- مؤشّر نحو المساحة التي نريد التلوين عليها (مثلا `screen`).
- الجزء من المساحة الذي نريد تلوينه، إذا أردت تلوين كل المساحة (و هذا الذّي نريده) فلتكن قيمة المؤشر `NULL`.
- اللون الذي نريد أن نلوّن به المساحة.

كملخّص :

```
SDL_FillRect(surface, NULL, color);
```

التحكم في الألوان بالـSDL

في الـ `SDL` كل لون مخزن في عدد من نوع `Uint32`.

> إذا كان عدداً، لماذا إذا لم نستعمل ببساطة النوع `int` أو النوع `long` ؟

الـ `SDL` هي مكتبة متعددة المنصات، و كما تعلم فحجم الـ `int` يتغير من نظام تشغيل إلى آخر. لهذا فإن الـ `SDL` تقوم باستخدام أعداد من أنواع جديدة، هذه الأنواع الجديدة تحجز نفس المكان بالذاكرة في كل أنظمة التشغيل.

هناك مثلاً :

- `Uint32` : عدد صحيح بحجم 32 bits أي 4 octets (للتذكير : 1 octet = 8 bits).
- `Uint16` : عدد صحيح مشفر على 16 bits (2 octets).
- `Uint8` : عدد طبيعي مشفر على 8 bits (1 octet).

لن تستعمل المكتبة سوى `typedef` لتقوم بتغيير قيمة العدد على حسب نظام التشغيل. إذا كنت فضولياً، فألق نظرة على الملف `SDL_types.h`.

لن تتأخر في التعامل مع كل هذا، فالتفاصيل لا تهم حالياً. كل ما عليك تذكره هو أن النوع `Uint32` لا يخزن إلا عدداً صحيحاً ليس إلا، مثل `int`.

> لكن كيف أعرف أي عدد يوافق اللون الذّي أريد ؟

هناك بالفعل دالة من أجل ذلك : `SDL_MapRGB`، هذه الأخيرة تستقبل 4 معاملات :

- صيغة الألوان : هذه الصيغة تعتمد على عدد bits/pixel التي قد طلبتها بالـ `SDL_SetVideoMode`. يمكنك استرجاع القيمة فهي موجودة في المتغير الداخلي `screen->format`.

- كمية الأحمر في اللون.
- كمية الأخضر في اللون.
- كمية الأزرق في اللون.

قد لا يعرف البعض بأن كل الألوان يتم تشكيلها عن طريق خلط الألوان : أزرق، أحمر و أخضر.
كل كمية تتدرج من العدد 0 (لا يوجد لون) إلى العدد 255 (كل اللون موجود). أي أننا لو كتبنا :

```
SDL_MapRGB(screen->format, 255, 0, 0)
```

فاللون المتشكل سيكون أحمرا. لا وجود للأخضر و لا للأزرق، أما لو نكتب :

```
SDL_MapRGB(screen->format, 0, 0, 255)
```

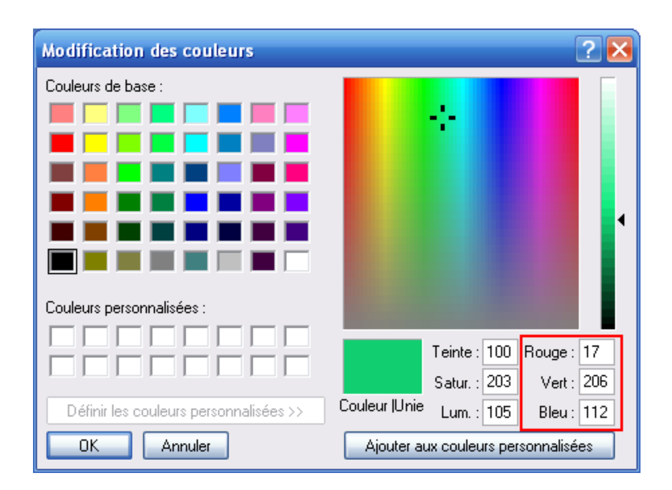
اللون سيكون أزرقا، بينما لو نكتب :

```
SDL_MapRGB(screen->format, 255, 255, 255)
```

اللون سيكون أبيضا لأننا دمجنا كل الألوان، لو أنك تريد تشكيل اللون الأسود، فلتجعل كل القيم على 0.

> ؟ ألا يمكننا استعمال لون آخر غير هذه الألوان ؟

كلّا، يمكنك ذلك لو أنك تقوم بمزج الألوان بشكل ذكي. للمساعدة في ذلك، توجه إلى برنامج Paint ثم إلى `Colors` / `Modify the colors`، انقر على `Define the colors` ثم `Custom`.
هنا، اختر اللون الذي يلائمك. أنظر إلى الصورة التالية :



مركّبات اللون متواجدة في أسفل يمين النافذة. كما ترى فقد اخترت لونا أخضر مزرقّا، و هو يتكوّن من 17 من الأحمر، 206 من الأخضر و 112 من الأزرق.

تلوين الشاشة

الدالة `SDL_MapRGB` تقوم بإرجاع عدد من نوع `Uint32` يوافق اللّون المختار.
يمكننا إذا تعريف متغير باسم `blueGreen` يحوي الشفرة الخاصة لاسترجاع هذا اللون :

```
Uint32 blueGreen = SDL_MapRGB(screen->format, 17, 206, 112);
```

ليس من الضروري المرور دائما على متغير لتخزين اللون المراد استعماله (إلا إن كنت تحتاجه فعلا في برنامجك).
يمكنك مباشرة إعطاء القيمة التي تم ارجاعها من طرف الدالة `SDL_MapRGB` إلى الدالة `SDL_FillRect`.

لو نريد أن نملأ الشاشة باللون الأخضر المزرق، يمكننا كتابة :

```
SDL_FillRect(screen , NULL, SDL_MapRGB(screen->format, 17, 206, 112));
```

لقد قمنا باستدعاء دالة خلال استدعاء دالة أخرى، أعتقد أنك تعرف بأن الأمر ممكن و لا يسبب أيّ مشاكل في لغة C.

تحديث الشاشة

لقد اقتربنا من تحقيق الهدف.
لقد نسينا أمراً بسيطاً : و هو الأمر بتحديث الشاشة. بالفعل، فالأمر `SDL_FillRect` يقوم بتلوين الشاشة، لكن هذا لا يحصل إلا في الذاكرة، إذ يجب أن نطلب من الحاسوب تحديث الشاشة لاستعمال البيانات الجديدة.

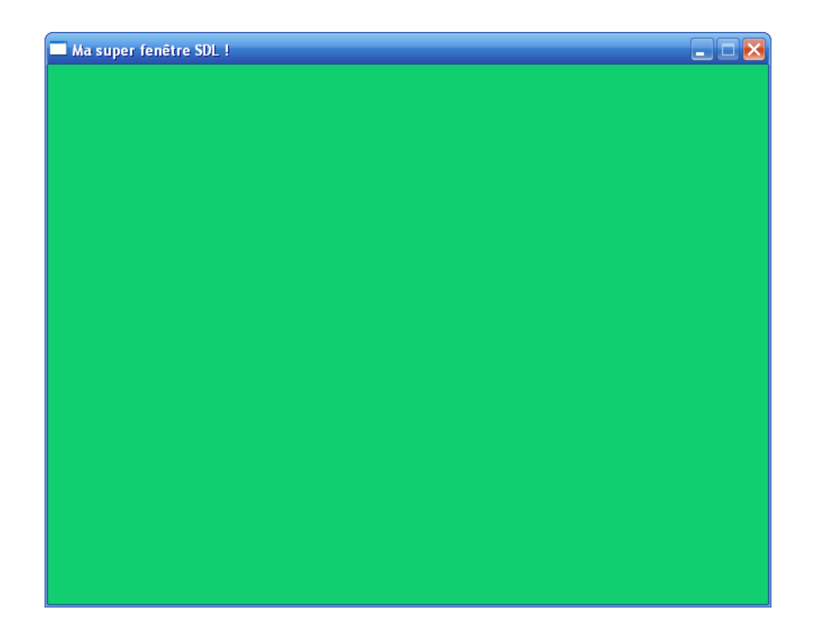
من أجل هذا سنستعمل الدالة `SDL_Flip`، سنتكلم بشكل مفصل عن هذه الدالة لاحقا.
الدالة تستقبل معاملا واحدا و هو الشاشة `screen`.

فلنلخّص كل شيء !

هذه دالة `main` تقوم بفتح نافذة ملونة باللون الأخضر المزرق :

```
int main(int argc, char *argv[])
{
        SDL_Surface *screen = NULL;
        SDL_Init(SDL_INIT_VIDEO);
        screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
        SDL_WM_SetCaption("Ma super fenêtre SDL !", NULL);
        // We colorize the screen with blue-green color
        SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 17, 206, 112));
        SDL_Flip(screen);
        pause();
        SDL_Quit();
        return EXIT_SUCCESS;
}
```

هاهي النتيجة :



رسم مساحة أخرى في الشاشة

النتيجة السابقة جيدة، لكننا لن نتوقف هنا. لحد الآن ليست لدينا سوى مساحة واحدة و هي الشاشة. نحن نريد أن نقوم بالرسم عليها، أي "نلصق" مساحات أخرى عليها بألوان مختلفة.

لهذا يجب علينا إنشاء متغير من نوع `SDL_Surface` للمساحة الجديدة :

```
1 SDL_Surface *rectangle = NULL;
```

سنطلب إذا من الـ`SDL` أن تقوم بحجز مكان في الذاكرة من أجل المساحة الجديدة.
من أجل الشاشة كنا قد استعملنا `SDL_SetVideoMode` . لكن هذه الأخيرة لا تعمل إلا على الشاشة (المساحة الرئيسية)، لا نريد أن نقوم بإنشاء نافذة من أجل كل مستطيل نريد إنشاءه !

توجد إذا دالة أخرى من أجل إنشاء مساحة : `SDL_CreateRGBSurface` . هذه هي التي سنقوم باستعمالها في كل مرة نريد أن ننشئ مساحة جديدة.

هذه الدالة تستقبل العديد من المعاملات (ثمانية !). لكنني لن أتطرّق إلا للمعاملات التي تهمّنا لحدّ الآن.
بما أن لغة C تُلزمنا بإدخال قيم لكل المعاملات، فإننا سنقوم بوضع القيمة 0 في مكان كل معامل لا يهمّنا.

فلنتأمل قليلا في المعاملات الأربع الأولى (يجدر بها أن تذكّرنا بإنشاء الشاشة).

- قائمة الأعلام (الخيارات). لديك الاختيار بين :
  - `SDL_HWSURFACE` : المساحة يتم تحميلها في الذاكرة الرسوميّة. و هي تحتوي على مكان أقل مقارنة بالذاكرة الخاصة بالنظام (حقيقة، مع بطاقات الـ3D في أيامنا هذه، قد لا يكون لهذا تأثير)، لكنها ذاكرات سريعة و فعّالة.

– `SDL_SWSURFACE` : يتم تحميل المساحة في الذاكرة الخاصة بالنظام، أين يوجد الكثير من المكان، لكن هذا الاختيار سيجبر المعالج على القيام بحسابات أكثر. لو أنك حمّلت المساحة على الذاكرة الرسوميّة، فإن البطاقة 3D هي المسؤولة عن القيام بأغلب الحسابات.

- عرض المساحة (pixels).
- ارتفاع المساحة (pixels).
- عدد الألوان (bits/pixel).

هكذا إذا نقوم بحجز مكان للمساحة الجديدة في الذاكرة :

```
rectangle = SDL_CreateRGBSurface(SDL_HWSURFACE, 220, 180, 32, 0, 0, 0, 0);
```

الأربع معاملات الأخيرة تساوي 0، كما قلت لك، لأننا لا نهتم بأمرها حالياً.

بما أننا قنا بالحجز اليدوي للذاكرة، فيجب علينا تحريرها باستعمال الدالة `SDL_FreeSurface` و التي نستعملها قبل `SDL_Quit` :

```
SDL_FreeSurface(rectangle);
SDL_Quit();
```

> ليس هناك من داعٍ إلى تحرير المساحة `screen` باستعمال `SDL_FreeSurface` لأنه يتم تحريرها تلقائياً عند استدعاء `SDL_Quit`.

يمكننا الآن تلوين المساحة الجديدة باللون الأبيض مثلا :

```
SDL_FillRect(rectangle, NULL, SDL_MapRGB(screen->format, 255, 255, 255));
```

لصق المساحة بالشاشة

اقتربنا من النهاية، هيا بعض الشجاعة ! المساحة جاهزة، لكن لو تحاول تجريب البرنامج، ستلاحظ أنها لن تظهر على الشاشة، بالفعل إذ أن المساحة `screen` هي وحدها التي تم إظهارها. لكي نستطيع رؤية مساحتنا الجديدة يجب أن نقوم بتسوية المساحة، أي لصقها على الشاشة، سنستعمل لأجل هذا الدالة `SDL_BlitSurface`. هذه الدالة تنتظر :

- المساحة التي نريد لصقها (هنا `rectangle`).
- معلومة حول الجزء من تلك المساحة الذي نريد لصقه (اختياري). لن يهمنا الأمر الآن فنحن نريد لصق كل المساحة و لهذا فستكون القيمة `NULL`.
- المساحة التي نريد أن نلصق عليها المساحة الجديدة (في حالتنا هذه نتكلم عن الشاشة `screen`).

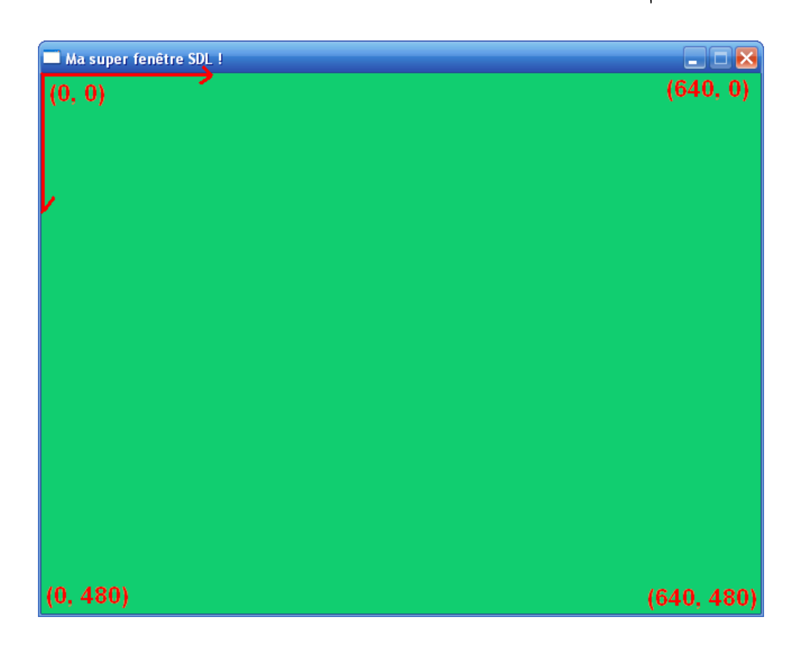
- مؤشّر نحو متغير يحتوي الإحداثيّات. هذه الإحداثيات تشير إلى المكان الذي نريد أن نلصق عليه المساحة، أي موقعه.

  للإشارة إلى الإحداثيّات، نحتاج إلى استعمال متغير من نوع `SDL_Rect`.
  إنّه هيكل يحتوي العديد من المركّبات، إثنتان منها تهمّنا :

  - `x` : الفاصلة.
  - `y` : الترتيبة.

يجب أن تعرف أن الإحداثيّة `(0, 0)` توافق أقصى نقطة في يسار أعلى الشاشة.
أما الإحداثيّة `(640, 480)` فهي توافق النقطة الموجودة في أقصى يمين أسفل الشاشة، و هذا إن كنت قد فتحت نافذة بحجم 640×480 مثلي.

هذا المخطط سيساعدك في الفهم :



إذا كنت قد درست الرياضيات من قبل، فعلى الأرجح لن تضيع بينما تحاول فهم كيفية العمل. فلننشئ إذا متغيرا `position`. سنعطي القيمة 0 لكل من الفاصلة و الترتيبة و ذلك ليتم لصق مساحتنا (المستطيل) في أعلى يسار النافذة :

```c
SDL_Rect position;
position.x = 0;
position.y = 0;
```

و الآن بما أننا حددنا موقعنا في النافذة، يمكننا تسوية المساحة الجديدة على الشاشة :

```c
SDL_BlitSurface(rectangle, NULL, screen, &position);
```

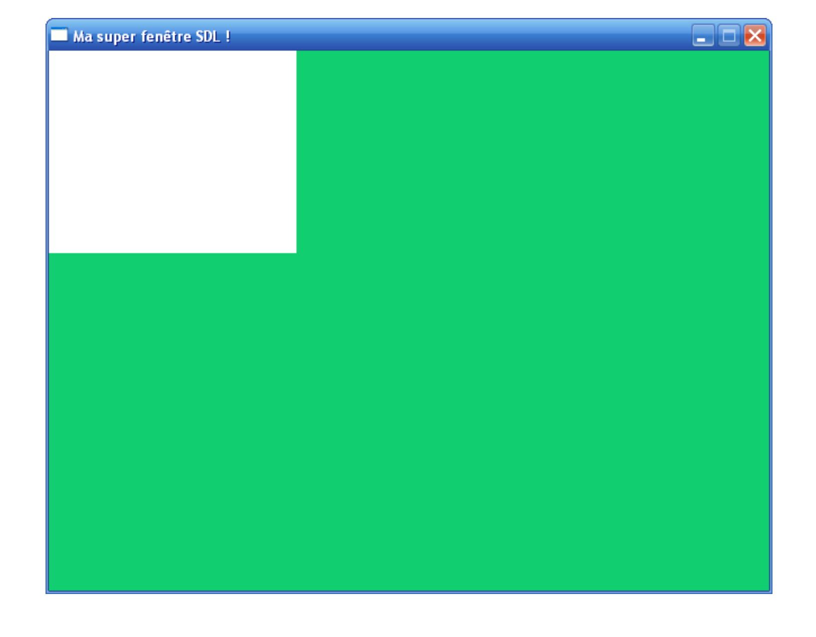
لاحظ أنني استعملت الرمز `&` و ذلك لأنه يجب علينا إرسال عنوان المتغير `position`.

تلخيص الشفرة المصدرية

أعتقد أن وضع الشفرة المصدرية الّتي تلخص ما شرحته لن يكون مضراً :

```
int main(int argc, char *argv[])
{
        SDL_Surface *screen = NULL, *rectangle = NULL;
        SDL_Rect position;
        SDL_Init(SDL_INIT_VIDEO);
        screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
        // Surface allocation
        rectangle = SDL_CreateRGBSurface(SDL_HWSURFACE, 220, 180, 32, 0,0, 0,
            0);
        SDL_WM_SetCaption("Ma super fenêtre SDL !", NULL);
        SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 17, 206,112));
        position.x = 0; // The coordinates of the surface will be (0, 0)
        position.y = 0;
        // Filling the surface with white color
        SDL_FillRect(rectangle, NULL, SDL_MapRGB(screen->format, 255,255, 255))
            ;
        SDL_BlitSurface(rectangle, NULL, screen, &position); // Sticking the
            surface on the screen
        SDL_Flip(screen); // Updating the screen
        pause();
        SDL_FreeSurface(rectangle); // Freeing the surface
        SDL_Quit();
        return EXIT_SUCCESS;
}
```

شاهد النتيجة :



مركزةُ المساحة في الشاشة

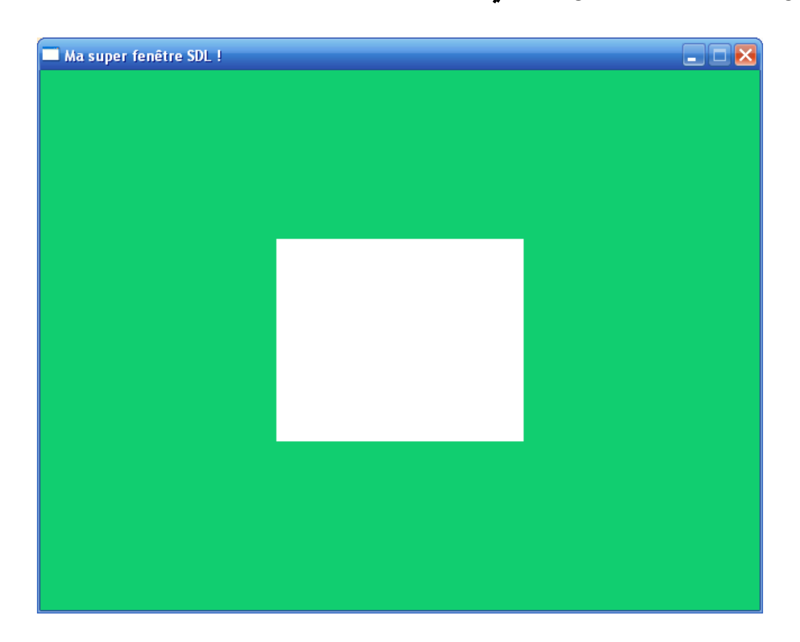
نحن نجيد إظهار المساحة في أعلى اليسار. يسهل أيضا موقعتها أسفل يمين الشاشة. ستكون الإحداثيات $(640 - 220, 480 - 180)$، لأنه يجب إنقاص حجم المستطيل ليتم إظهاره كاملا.

لكن كيف تتم مركزةُ المستطيل الأبيض ؟ لو تفكّر قليلاً ستجد بأن الحساب رياضياتيّ. فهنا نعرف الهدف من الرياضيات و الحساب الهندسي !
كلّ هذا الأمر بمستوى سهل هنا :

```
position.x = (640 / 2) — (220 / 2);
position.y = (480 / 2) — (180 / 2);
```

فاصلة المستطيل هي نصف عرض الشاشة $(640/2)$. و لكن، بالإضافة إلى هذا، يجب أن يتم إنقاص نصف طول المستطيل أيضاً $(220/2)$، لأنك إن لم تنقص هذا الحجم، سيكون تمركز المستطيل خاطئاً (جرّب عدم فعل ذلك و ستفهم ما الّذي أعنيه).
كذلك بالنسبة للترتيبة مع ارتفاع الشاشة و المستطيل.

النتيجة : المستطيل الأبيض يتركز بشكل جيد في الشاشة.



4.21 تمرين : إنشاء تدرّج لونيّ

سننهي الفصل بتمرين صغير (مصحّح) متبوع بسلسلة تمارين أخرى (غير مصححة من أجل حثّك على التدريب).

التمرين المصحح ليس صعباً حقّا : ما نريد إنشاءه هو نافذة متدرّجة الألوان عموديا من الأسود إلى الأبيض.
سيكون عليك إنشاء 255 مساحة بارتفاع 1 بيكسل. كل مساحة لها لون مختلف أكثر فأكثر سوادا.

هذا ما يجب عليك الحصول عليه في النهاية، صورة مشابهة لهذه :



إنه أمر جميل، أليس كذلك ؟ الشيء الأجمل هو أن بعض الحلقات التكرارية كافية من أجل تحقيق المطلوب.

لفعل ذلك يجب إنشاء 256 مساحة (أي 256 سطر) تحتوي مركبات الألوان (أحمر، أخضر، أزرق) التالية :

```
0, 0, 0) // Black
(1, 1, 1) // Gray that is so so close from black
(2, 2, 2) // Gray that is so close from black
...
(128, 128, 128) // Medium gray (to 50 %)
...
(253, 253, 253) // Gray that is so  close from white
(254, 254, 254) // Gray that is so so close from white
(255, 255, 255) // White
```

يجب على أيّ كان أن يعرف أنّه بحاجة إلى حلقة تكراريّة للقيام بهذا (لن تسعد بتكرار 256 سطرا !). و لهذا سنقوم بإنشاء جدول من نوع `SDL_Surface*` من 256 خانة.

إلى العمل. لديك 5 دقائق !

تصحيح !

يجب أولا أن نقوم بتعريف جدول من 256 `SDL_Surface*`. سنهيّؤه على `NULL` :

```
SDL_Surface *lines[256] = {NULL};
```

سنعرف متغيراً `i` من أجل الحلقات `for`.

سنغيّر أيضاً ارتفاع النافذة لكي تكون مناسبة للعمل. إذ سنعطيها 256 بيكسلز كارتفاع، و ذلك من أجل عرض كل سطر من بين 256 سطرا.

سنستعمل بعد ذلك حلقة تكرارية `for` من أجل حجز مكان لـ256 مساحة التّي تم إنشاؤها. الجدول سيستقبل 256 مؤشّرا إلى كلّ واحد من المساحات المنشأة :

```
for (i = 0 ; i <= 255 ; i++)
        lines[i] = SDL_CreateRGBSurface(SDL_HWSURFACE, 640, 1, 32, 0,0, 0, 0);
```

بعد ذلك نقوم بملء و لصق كل مساحة في الشاشة واحدة بواحدة.

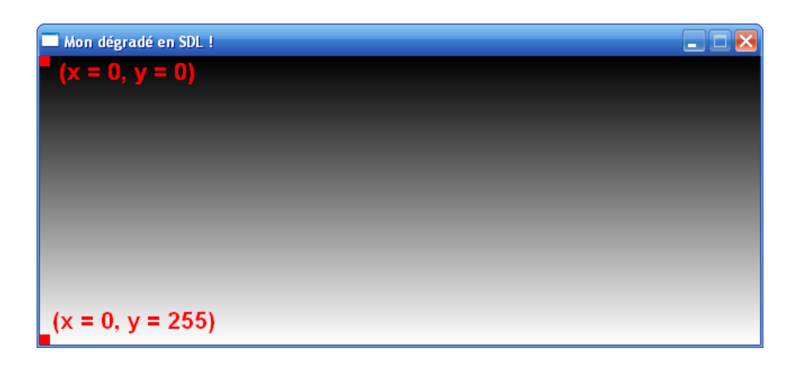
```
for (i = 0 ; i <= 255 ; i++)
{
        position.x = 0; // The lines are to the left (0 abscissa)
        position.y = i; // The vertical position depends on the line's number
        SDL_FillRect(lines[i], NULL, SDL_MapRGB(screen->format, i, i, i)); //
            Drawing
        SDL_BlitSurface(lines[i], NULL, screen, &position); // Sticking
}
```

لاحظ أنني استعمل كل الوقت المتغير `position` . إذ ليس لازما أن ننشئ 256 واحدا، لأننا لن نقوم إلا ببعث المتغير إلى الدالة `SDL_BlitSurface` . يمكننا إذن إعادة استخدامه دون مشاكل.
في كلّ مرة أقوم بالتعديل على الترتيبة (`y`)، لتسوية المساحة على الارتفاع الصحيح. اللون يعتمد في كلّ مرة على قيمة المتغير `i` (ستكون $0, 0, 0$ في أوّل مرّة و $255, 255, 255$ في آخر مرّة).

> ؟
>
> لكن لماذا قيمة `x` هي 0 دائماً ؟
> كيف يمكن للمساحة أن تلون كليا إذا كانت قيمة الـ `x` دائما 0 ؟

المتغير `position` يشير إلى أي مكان تتواجد فيه المنطقة أعلى اليسار (هنا نتكلم عن السطر). هي لا تحدد عرض المساحة و إنما فقط أين تتواجد المركّبة على الشاشة.
بما أن كل الأسطر تبدأ في أقصى يسار النافذة، فستكون الفاصلة مساوية لـ0. حاول وضع فاصلة تساوي 50 لترى ماذا سيعطيك : كل الأسطر ستنتحي إلى اليمين.
بما أن المساحة تأخذ 640 بيكسل كطول ، فإن الـSDL تقوم بإنشاء 640 بيكسلا في اتجاه اليمين (من نفس اللون) إنطلاقاً من المركبات التي يشير إليها المتغير `position` .

في المخطط التالي أريك مركّبات النقطة المتواجدة أعلى يسار الشاشة (وضعية أول سطر) ومركبات النقطة المتواجدة أسفل يسار الشاشة (وضعية آخر سطر).



كما ترى، من الأعلى إلى الأسفل، المحور لا يتغير (`x` يبقى مساويا لـ0) بينما `y` وحده يتغير من أجل كل سطر جديد، و بهذا `position.y = i;` .

أخيراً لا تنس أنه يجب تحرير الذاكرة من أجل كل مساحة من الـ256 مساحة المنشأة، بمساعدة حلقة بالطبع.

```c
for (i = 0 ; i <= 255 ; i++) // Don't forget to free the 256 surfaces
        SDL_FreeSurface(lines[i]);
```

ملخّص main

هذه هي الدالة `main` كاملة :

```c
int main(int argc, char *argv[])
{
        SDL_Surface *screen = NULL, *lines[256] = {NULL};
        SDL_Rect position;
        int i = 0;
        SDL_Init(SDL_INIT_VIDEO);
        screen = SDL_SetVideoMode(640, 256, 32, SDL_HWSURFACE);
        for (i = 0 ; i <= 255 ; i++)
                lines[i] = SDL_CreateRGBSurface(SDL_HWSURFACE, 640, 1, 32,0, 0,
                    0, 0);
        SDL_WM_SetCaption("Mon dégradé en SDL !", NULL);
        SDL_FillRect(screen , NULL, SDL_MapRGB(screen ->format, 0, 0, 0));
        for (i = 0 ; i <= 255 ; i++)
        {
                position.x = 0; // The lines are to the left
                position.y = i; // The vertical position depends on the line's
                    number
                SDL_FillRect(lines[i], NULL, SDL_MapRGB(screen->format, i, i, i
                    ));
                SDL_BlitSurface(lines[i], NULL, screen, &position);
        }
        SDL_Flip(screen);
        pause();
        for (i = 0 ; i <= 255 ; i++) // Don't forget to free the 256 surfaces
                SDL_FreeSurface(lines[i]);
        SDL_Quit();
        return EXIT_SUCCESS;
}
```

"أريد تمارين للتدريب !"

لا مشكلة، مولّد التمارين مُشَغّل !

- قم بإنشاء تدرج عكسي للألوان، أي من الأبيض للأسود. هذا الأمر لن يكون صعبا للبدأ !
- يمكنك أيضاً وضع كلى التدرّجين، من الأبيض للأسود ثم من الأسود للأبيض (ستأخذ النافذة ضعف الارتفاع الحالي).
- أكثر صعوبة قليلا، يمكنك وضع تدرج أفقي بدل التدرج العمودي.

- حاول إنشاء تدرج ألوان مختلفة عن الأسود و الأبيض. جرب مثلا من الأحمر إلى الأسود، من الأخضر إلى الأسود، و من الأزرق إلى الأسود، ثمّ من الأحمر إلى الأبيض، إلخ.

ملخّص

- يتم تحميل الـ `SDL` بواسطة الـ `SDL_Init` في بداية البرنامج، و يتم إيقافها باستعمال `SDL_Quit` في النهاية.
- الأعلام هي ثوابت يمكن جمعها فيما بينها باستعمال الرمز `|`، و هي تلعب دور الخواص.
- تقوم الـSDL بالتعامل مع المساحات و التي هي عبارة عن مستطيلات من نوع `SDL_Surface`. الرسم على النافذة يتم بالاستعانة بهذه المساحات.
- توجد دائما على الأقل مساحة واحدة و التي تحجز كلّ النافذة، و نسميها في أغلب الأحيان الشاشة (`screen`).
- ملء مساحة يتم باستعمال `SDL_FillRect`، ولصقها في الشاشة يتم باستعمال `SDL_BlitSurface`.
- الألوان معرّفة بمزيج من الأحمر، الأزرق و الأخضر.

الفصل 22

إظهار صور

لقد تعلّمنا كيف نقوم بتحميل الـSDL، فتح نافذة و التعامل مع المساحات. إنها بالفعل من المبادئ التي تجب معرفتها عن هذه المكتبة. لكن لحدّ الآن لا يمكننا سوى إنشاء مساحات موحّدة اللون، و هذا الأمر بدائي قليلاً.

في هذا الفصل، سنتعلّم كيف نقوم بتحميل صور على مساحات، مهما كانت صيغتها BMP، PNG أو حتى GIF أو JPG. التحكم في الصور أمر مهم للغاية لأنه بتجميع الصور (نسميها أيضاً "sprites") نضع اللبنات الأولى في بناء لعبة فيديو.

1.22 تحميل صورة BMP

الـSDL هي مكتبة بسيطة جداً. فهي لا تستطيع أساسا تحميل سوى صور من نوع "bitmap" (ذات امتداد `.bmp`).
لا تقلق، فبفضل إضافة خاصّة بالـSDL (المكتبة SDL_Image)، سنرى بأنه بإمكاننا أيضاً تحميل صور من صيغ أخرى.

للبدأ، سنكتفي الآن بما تسمح لنا به الـSDL بشكل قاعدي. سنقوم بدراسة تحميل صور BMP.

الصيغة BMP

الصيغة BMP (اختصار لـbitmap) هي صيغة صور.
الصور الّتي نجدها في الحاسوب مخزّنة في ملفات. يوجد العديد من صيغ الصور، أي العديد من الطرق لتخزين صورة في ملف. على حسب الصيغة، يمكن للصورة أخذ الكثير أو القليل من مساحة القرص الصلب، و تملك جودة أحسن أو أسوء.

الـBitmap هي صيغة غير مضغوطة (على عكس الـJPG، PNG، GIF، إلخ). فعليّا، هذا يعني الأمور التالية :

- يكون الملف سريعاً جداً من ناحية قراءته، على عكس الصيغ المضغوطة التي يجب أن يتم فك الضغط عنها، مما يكلّفنا بعض الوقت.
- جودة الصورة مثالية. بعض الصيغ المضغوطة (أفكّر في الـJPG خصوصا، لأن الـPNG و الـGIF لا يغيّرون في الصورة) تقوم بتخريب جودة الصورة، و هذا ليس هو الحال بالنسبة للـBMP.
- لكنّ الملف سيكون ضخماً بما أنه ليس مضغوطاً !

توجد هناك إذا مزايا و مساوئ.
بالنسبة للـSDL، الشيء الجيد هو أن نوع الملف سيكون بسيطا و سهل القراءة. إذا كان عليك تحميل الصور دائمًا في نفس وقت تشغيل برنامجك، من المستحسن استعمال صور بصيغة BMP. سيكون حجم الملف ضخما حتما، لكنه يُحمّل بشكل أسرع من الـGIF مثلاً. سيكون الأمر مهّما إذا كان على برنامجك تحميل الكثير من الصور في وقت قصير.

تحميل صورة Bitmap

تنزيل حزمة الصور

في هذا الفصل سنقوم بالعمل على كثير من الصور. إذا أردت القيام بتجريب الشفرات بينما أنت تقرأ (و هذا ما يجدر بك فعله !)، فأنصحك بتنزيل حزمة الصور التي تحتوي كل الصور التي نحتاج إليها.

https://openclassrooms.com/uploads/fr/ftp/mateo21/pack_images_sdz.zip (1 Mo)

بالطبع، يمكنك استعمال صورك الخاصة. يجب عليك فقط أن تعدّل مقاييس النافذة على حسب مقاييس الصورة.

قم بوضع كل الصور في مجلّد المشروع. سنبدأ أولاّ بالعمل على الصورة `lac_en_montagne.bmp` . هي عبارة عن لقطة تم استخلاصها من مشهد ثلاثي الأبعاد مأخوذ من البرنامج الممتاز الخاص بنمذجة المناظر الطبيعية Vue d'Espri 4، و الذي تم إيقاف تسويقه. منذ ذلك، تمّ تغيير اسم البرنامج إلى Vue و تم تطويره كثيراً. لمن يريد معرفة المزيد عنه، يمكنه زيارة الموقع :
http://www.e-onsoftware.com/.

تحميل صورة في مساحة

سنقوم باستعمال دالّة تقوم بتحميل صورة ذات صيغة BMP و لصقها في مساحة.
هذه الدالة تدعى `SDL_LoadBMP` و سترى أن استعمالها سهل للغاية :

```
1 mySurface = SDL_LoadBMP("image.bmp");
```

الدالة `SDL_LoadBMP` تقوم بتعويض دالتين تعرفهما :

- `SDL_CreateRGBSurface` : تقوم بحجز مكان في الذاكرة من أجل تخزين مساحة ذات الحجم المطلوب (تكافئ دالة `malloc`).
- `SDL_FillRect` : تقوم بملئ الهيكل بلون موحّد.

لماذا تقوم الدالة بتعويض هذين الدالتين ؟ الأمر بسيط :

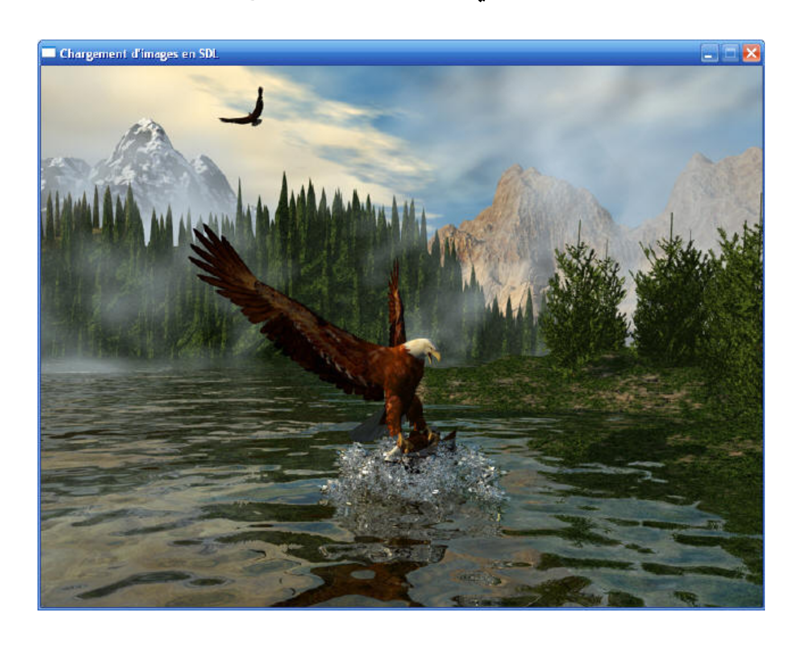
- الحجم الذي نقوم بحجزه في الذاكرة من أجل المساحة يعتمد على حجم الصورة : اذا كان حجم الصورة هو 250×300 فستأخذ المساحة نفس الحجم.

- من جهة أخرى، يتم ملأ المساحة بيكسلا ببيكسل بمحتوى الصورة BMP.

فلنكتب الشفرة دون أي تأخير :

```
1  int main(int argc, char ▫argv[])
2  {
3          SDL_Surface ▫screen = NULL, ▫backgroundImage = NULL;
4          SDL_Rect backgroundPosition;
5          backgroundPosition.x = 0;
6          backgroundPosition.y = 0;
7          SDL_Init(SDL_INIT_VIDEO);
8          screen = SDL_SetVideoMode(800, 600, 32, SDL_HWSURFACE);
9          SDL_WM_SetCaption("Chargement d'images en SDL", NULL);
10         /□ Loading a Bitmap image in a surface □/
11         backgroundImage = SDL_LoadBMP("lac_en_montagne.bmp");
12         /□ We blit on the screen □/
13         SDL_BlitSurface(backgroundImage, NULL, ecran, &backgroundPosition);
14         SDL_Flip(screen);
15         pause();
16         SDL_FreeSurface(backgroundImage); // We free the surface
17         SDL_Quit();
18         return EXIT_SUCCESS;
19 }
```

و بهذا أكون قد أنشأت مؤشّراً نحو مساحة (`backgroundImage`) و نحو كل المركّبات الموافقة لها (`backgroundPosition`).
تم إنشاء المساحة في الذاكرة و ملؤها من طرف الدالة `SDL_LoadBMP` .
نقوم بتسويتها على المساحة `screen` و هذا كلّ شيء ! الصورة التالية توضّح النتيجة :



كما ترى، لم يكن الأمر صعباً !

إرفاق أيقونة بالتطبيق

بما أننا الآن نجيد تحميل الصور، يمكننا اكتشاف كيفية إرفاق أيقونة بالبرنامج. سيتم إظهار الأيقونة في أعلى يسار النافذة (و أيضاً في شريط المهام). لحدّ الآن نحن لا نملك إلاّ أيقونة افتراضيّة.

؟ لكن ألا يجدر بأيقونات البرامج أن تكون ذات الامتداد `.ico` ؟

كلّا، ليس شرطاً ! على كلّ فالامتداد `.ico` لا يوجد إلا في نظام Windows. الـSDL تتعامل مع كلّ أنظمة التشغيل باستعمالها نظاما خاصا بها : المساحة !
نعم، أيقونة برنامج SDL ماهي إلا مساحة بسيطة.

! يجدر بالأيقونة أن تكون ذات حجم 16×16 بيكسلز. بينما في Windows يجب أن تكون بحجم 32×32 بيكسلز و إلا فستسوء جودتها. لا تقلق إذ يمكن للـSDL "تصغير" أبعاد الصورة لتتمكن من الدخول في 16×16 بيكسلز.

لإضافة الأيقونة إلى النافذة، نستعمل الدالة `SDL_WM_SetIcon`.
هذه الدالة تأخذ معاملين : المساحة التي تحتوي الصورة التي نريد إظهارها كما أنها تستقبل معلومات حول الشفافية (القيمة `NULL` تعني أننا لا نريد أية شفافية). التحكّم في الشفافية الخاصة بأيقونة معقّد قليلاً (يجب تحديد البيكسلز الشفافة واحدا بواحد)، لن ندرس ذلك إذاً.

سنقوم باستدعاء دالة في استدعاء لأخرى :

```
1 SDL_WM_SetIcon(SDL_LoadBMP("sdl_icone.bmp"), NULL);
```

تم تحميل الصورة في الذاكرة بواسطة `SDL_LoadBMP` و بعث عنوان المساحة مباشرة إلى `SDL_WM_SetIcon`.

X يجب أن يتم استدعاء الدالة `SDL_WM_SetIcon` قبل أن يتم فتح النافذة، أي أنه يجدر بها التواجد قبل `SDL_SetVideoMode` في الشفرة المصدرية.

هذه هي الشفرة المصدرية الكاملة. ستلاحظ أنني أضفت `SDL_WM_SetIcon` مقارنة بالشفرة السابقة.

```
1  int main(int argc, char *argv[])
2  {
3          SDL_Surface *screen = NULL, *backgroundImage = NULL;
4          SDL_Rect backgroundPosition;
5          backgroundPosition.x = 0;
6          backgroundPosition.y = 0;
7          SDL_Init(SDL_INIT_VIDEO);
8          /* Loading the icon before SDL_SetVideoMode*/
9          SDL_WM_SetIcon(SDL_LoadBMP("sdl_icone.bmp"), NULL);
10         screen = SDL_SetVideoMode(800, 600, 32, SDL_HWSURFACE);
```

```
11        SDL_WM_SetCaption("Chargement d'images en SDL", NULL);
12        backgroundImage = SDL_LoadBMP("lac_en_montagne.bmp");
13        SDL_BlitSurface(backgroundImage, NULL, screen, &backgroundPosition);
14        SDL_Flip(screen);
15        pause();
16        SDL_FreeSurface(backgroundImage);
17        SDL_Quit();
18        return EXIT_SUCCESS;
19 }
```

النتيجة : تم تحميل الصورة و عرضها أعلى يسار النافذة.



2.22 التحكم في الشفافية

مشكل الشفافية

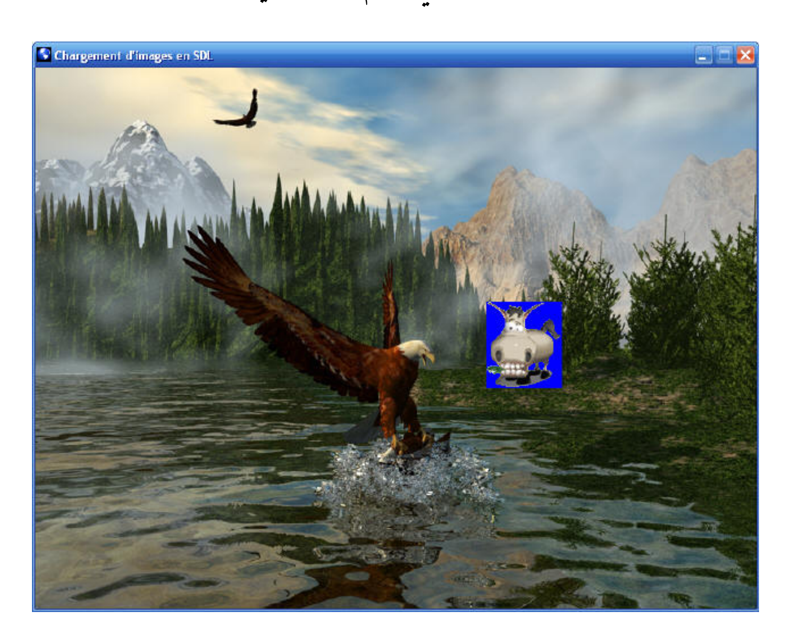
لقد قمنا قبل قليل بتحميل صورة bitmap في النافذة.
لنفرض أننا نريد لصق صورة فوقها. و هذا ما يحصل كثيراً في الألعاب. غالبا اللاعب الذي يتحرّك في الخريطة هو عبارة عن صورة bitmap تتحرك فوق صورة خلفية.

سنقوم بلصق صورة Zozor (لمن لا يعرفه، فهو شعار Site du Zéro سلف الموقع OpenClassrooms حاليًا) في المشهد :

```
1  int main(int argc, char *argv[])
2  {
3        SDL_Surface *screen = NULL, *backgroundImage = NULL, *zozor = NULL;
4        SDL_Rect backgroundPosition, zozorPosition;
5        backgroundPosition.x = 0;
6        backgroundPosition.y = 0;
7        zozorPosition.x = 500;
8        zozorPosition.y = 260;
9        SDL_Init(SDL_INIT_VIDEO);
10       SDL_WM_SetIcon(SDL_LoadBMP("sdl_icone.bmp"), NULL);
11       screen = SDL_SetVideoMode(800, 600, 32, SDL_HWSURFACE);
12       SDL_WM_SetCaption("Chargement d'images en SDL", NULL);
13       backgroundImage = SDL_LoadBMP("lac_en_montagne.bmp");
14       SDL_BlitSurface(backgroundImage, NULL, ecran, &backgroundPosition);
15       // Loading and blitting Zozor on the screen
16       zozor = SDL_LoadBMP("zozor.bmp");
17       SDL_BlitSurface(zozor, NULL, screen, &zozorPosition);
18       SDL_Flip(screen);
19       pause();
20       SDL_FreeSurface(backgroundImage);
```

```
21        SDL_FreeSurface(zozor);
22        SDL_Quit();
23        return EXIT_SUCCESS;
24    }
```

لقد قمنا فقط بإضافة مساحة لنخزّن فيها Zozor، و التي نقوم بلصقها في مكان معيّن من المشهد :



يبدو المشهد سيّئا، أليس كذلك ؟

> ؟ بالطبع يعود ذلك إلى الخلفية الزرقاء التي هي خلف Zozor !

لأنك تعتقد أنه بوجود خلفية سوداء أو بنّيّة، ربما سيكون المظهر لائقاً أكثر ؟ لا بالطبع، المشكل هنا هو أنه من اللازم أن يكون شكل الصورة عبارة عن مستطيل، أي أنه إذا قمنا بلصقها على المشهد، سنرى خلفيتها، مما يشوّه المظهر.

من حسن الحظّ أن الـSDL تتحكّم في الشفافية !

جعل صورة شفافة

الخطوة 1 : تحضير الصورة

كبداية، يجب تحضير الصورة التي نريد تسويتها على المشهد.
الصيغة BMP لا تتحكم في الشفافية، على عكس الصيغتين GIF و PNG. لهذا يجب علينا أن نجد حلاً آخر.

يجب استعمال نفس اللون للخلفية على الصورة. هذه الأخيرة ستكون شفافة من طرف الـSDL في وقت التسوية.
لاحظ كيف تبدو الصورة `zozor.bmp` من ناحية أقرب :



الخلفية الزرقاء إذا مُختارة. لاحظ أنني اخترت اللون الأزرق بشكل عشوائي، كان بإمكاني استعمال اللون الأحمر أو الأصفر مثلاً. الشيء المهم هو أنه يجب على اللون أن يكون وحيداً و موحّدا. لقد اخترت اللون الأزرق لأنه ليس متواجدا في صورة Zozor لأنني لو اخترت اللون الأخضر، سأخاطر بجعل العشب الذي يتناوله الحمار (أسفل يسار الصورة) شفافاً.

استعمل إذا أي برنامج كان (Paint، Photoshop، The Gimp، ... لكلّ واحد منّا ذوقه) لإعطاء خلفية موحّدة للصورة.

الخطوة 2 : تحديد اللون الشفاف

لكي نقوم بتحديد اللون الذي يجب أن تجعله SDL شفافاً، يجب أن نستعمل الدالة `SDL_SetColorKey` . يجب استدعاء هذه الدالة قبل تسوية الصورة.
هكذا نقوم بتحويل اللون الذي خلف Zozor إلى الشفاف :

```
SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));
```

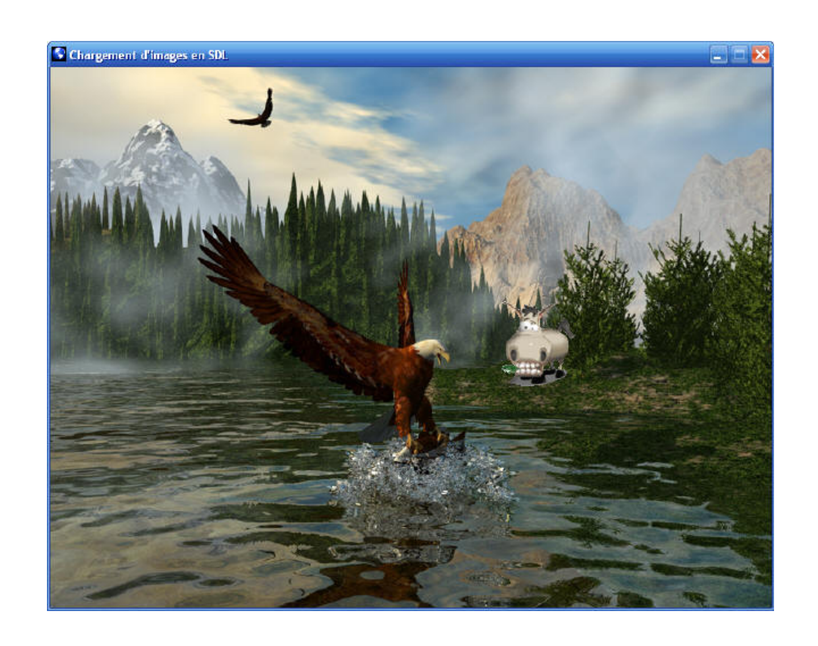
هناك ثلاثة معاملات :

- المساحة التي يجب أن نقوم بتحويلها إلى اللون الشفاف (هنا نتكلم عن `zozor`).
- قائمة الأعلام : استعمل `SDL_SRCCOLORKEY` لتفعيل الشفافية، 0 من أجل تعطيلها.
- حدد بعد ذلك اللون الذي يجب أن يتم تحويله إلى الشفاف. لقد استعملت `SDL_MapRGB` لإنشاء اللون بصيغة عدد (`Uint32`) كما فعلنا بالسابق. كما ترى إنه اللون الأزرق $(0, 0, 255)$ الّذي سيتم تحويله إلى الشفاف.

كملخص، نقوم أولا بتحميل الصورة باستعمال `SDL_LoadBMP`، ثمّ نحدد اللون الشفاف باستعمال `SDL_SetColorKey` ثم نقوم بتسوية المساحة باستعمال `SDL_BlitSurface`.

```
zozor = SDL_LoadBMP("zozor.bmp");
SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));
SDL_BlitSurface(zozor, NULL, screen, &zozorPosition);
```

النتيجة : تم دمج صورة Zozor بشكل ممتاز في المشهد :



هذه هي التقنية المبدئية التي ستعيد استعمالها كل الوقت في برامجك. تعلّم كيف تحكم جيداً بالشفافية لأنها من أساسيات صنع لعبة تملك الحدّ الأدنى من الواقعية.

الشفافية Alpha

هو نوع آخر من الشفافية.
لحدّ الآن قمنا بتعريف لون واحد شفاف (الأزرق مثلا). هذا اللون لا يظهر في الصورة الملُصقة.

الشفافية Alpha توافق شيئاً آخر، إنها تسمح بعمل "مزج" بين صورة و خلفية. هذا نوع من التلاشي.

يمكن تفعيل الشفافية Alpha لمساحة عن طريق الدالة `SDL_SetAlpha` :

```
SDL_SetAlpha(zozor, SDL_SRCALPHA, 128);
```

يوجد هنا ثلاثة معاملات كذلك :

- المساحة التي نتكلم عنها (`zozor`).
- قائمة الأعلام : ضع `SDL_SRCALPHA` من أجل تفعيل الشفافية، 0 من أجل تعطيلها.
- مهم جدا : قيمة الشفافية Alpha هي عدد يتراوح بين 0 (صورة شفافة تماماً أي غير مرئية) و 255 (صورة ظاهرة كلياً، و كأن الشفافية Alpha لم تكن موجودة).

كلما كان العدد Alpha صغيراً كلما زادت شفافية الصورة و تلاشيها في الخلفية.

هذا مثال عن شفرة تقوم بتطبيق شفافية بقيمة 128 على الصورة Zozor :

```
zozor = SDL_LoadBMP("zozor.bmp");
SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));
/* Average Alpha transparency (128) : */
SDL_SetAlpha(zozor, SDL_SRCALPHA, 128);
SDL_BlitSurface(zozor, NULL, screen, &zozorPosition);
```

تلاحظ أنني حافظت على شفافية `SDL_SetColorKey` . يمكن دمج النوعين الاثنين للشفافية معاً.

الجدول التالي يوضّح لك كيف يبدو Zozor باختلاف قيم Alpha.



Alpha	النتيجة
255 (مرئيّة بالكامل)	
190	
128 (شفافيّة متوسّطة)	
75	
0 (غير مرئيّة بالكامل)	

م
قيمة الشفافية Alpha 128 (شفافية متوسطة) هي قيمة خاصّة و كثيرة الإستعمال بالـSDL. هذا النمط من الشفافية أسرع من ناحية حسابات المعالج مقارنة بالأنماط الأخرى. قد يكون من المهم لك معرفة هذه المعلومة خاصة إن كنت تستعمل الشفافية Alpha بشكل كبير في برامجك.

3.22 تحميل صيغ صور أخرى باستعمال الـSDL_Image

الـSDL لا تتعامل إلا مع الـbitmap (الصيغة BMP) كما رأينا.
و لكن هذا ليس بمشكل لأن قراءة الصور ذات الصيغة BMP أسرع بالنسبة للـSDL، و لكن يجب معرفة أنه في أيامنا هذه يتم استعمال صيغ أخرى للصور. بالتحديد الصيغ "المضغوطة" كالـPNG، الـGIF و الـJPEG. لهذا الغرض توجد مكتبة تسمى SDL_Image و تقوم بالتعامل مع كل صيغ الصور التالية :

- TGA،
- BMP،
- PNM،
- XPM،
- XCF،
- PCX،
- GIF،
- JPG،
- TIF،
- LBM،
- PNG.

بالمناسبة فإنه بالإمكان أن تتم إضافة صيغ أخرى للـSDL. و هي المكتبات التي تحتاج إلى الـSDL لكي تعمل. يمكننا تسمية هذا الأمر بالـadd-ons (بمعنى "إضافات"). SDL_Image هي واحدة من بين هذه المكتبات.

تثبيت الـSDL_Image على Windows

التنزيل

توجد صفحة خاصة من موقع الـSDL تشير إلى المكتبات التي تستعملها الـSDL. هذه الصفحة تحمل عنوان "Libraries". ستجد رابطاً في القائمة اليسارية.
ستلاحظ أن هناك الكثير من المكتبات و أغلبها ليس من طرف المبرمجين الأصليين للـSDL. بل هم مبرمجون عاديون يستعملون الـSDL و يقومون باقتراح مكتباتهم الخاصة لتحسين هذه الأخيرة.

بعض هذه المكتبات مفيد جداً و يستحق إلقاء النظر عليه، و بعضها أقلّ جودة بل ربّما فيه أخطاء. لهذا يجب ترتيب هذه المكتبات حسب أهميتها.

حاول إيجاد SDL_Image في القائمة، ستدخل إلى الصفحة المخصصة لهذه المكتبة :

https://www.libsdl.org/projects/SDL_image

نزّل النسخة التي تناسبك من القسم "Binary" (لا تحمّل الملفات المصدرية، لن تحتاجها !).

إذا كنت تعمل على Windows، نزّل الملف `SDL_image-devel-1.2.10-VC.zip`، و هذا حتى و إن لم تكن تستعمل البيئة التطويرية ++Visual C !

التثبيت

في الملف `.zip` هذا، ستجد :

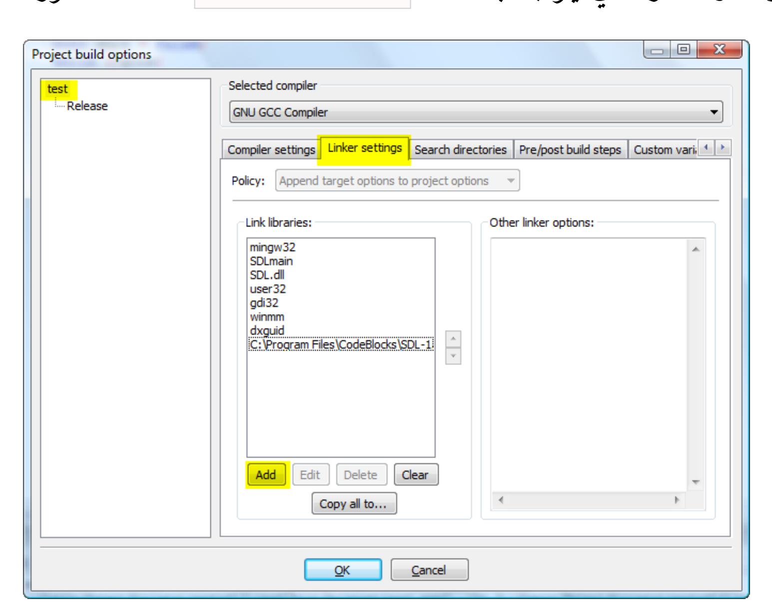
- `SDL_image.h` : الملف الرأسي الوحيد الذي تحتاجه الـSDL_Image، قم بلصقه في المسار `C:\Program Files\CodeBlocks\SDL-1.2.13\include`
  بمعنى آخر، إلى جانب الملفات الرأسية للـSDL.
- `SDL_image.lib` : قم بلصقه في المسار
  `C:\Program Files\CodeBlocks\SDL-1.2.13\lib`.

أعرف أنك ستخبرني بأن الملفات ذات الامتداد `.lib` هي محجوزة للبيئة التطويرية ++Visual C، لكن هذه حالة استثنائية، فالملف `.lib` يعمل حتى مع المترجم mingw.

- الكثير من الملفات DLL : قم بوضعها كلها في المجلّد الخاص بالمشروع (أي بجانب الملف `SDL.dll`).

بعد ذلك، يجدر بك تغيير خواص المشروع من أجل محرّر الروابط (Linker) للملف `SDL_image.lib`.

إذا كنت تعمل بالـCode::Blocks مثلاً، توجّه إلى القائمة `Projects` / `Build options`، في الفرع `Linker` أنقر على الزر `Add` و اختر المسار الذي يتواجد به الملف `SDL_image.lib`، لاحظ الصورة :



إذا ظهرت لك رسالة تحمل سؤال : *”Keep as a relative path ?”*، فلتجب بما أردت لأنه لن يغير شيئاً في الوقت الحالي. أنصحك بالإجابة بالسلب، شخصيّاً.

بعد ذلك، ما عليك سوى تضمين الملف الرأسي `SDL_image.h` في الشفرة المصدرية. على حسب المكان الذي وضعت فيه الملف `SDL_image.h` سيكون عليك استعمال هذه الشفرة :

```
#include <SDL/SDL_image.h>
```

أو هذه

```
#include <SDL_image.h>
```

جرّبهما كليهما، يجدر بأحداهما أن تعمل.

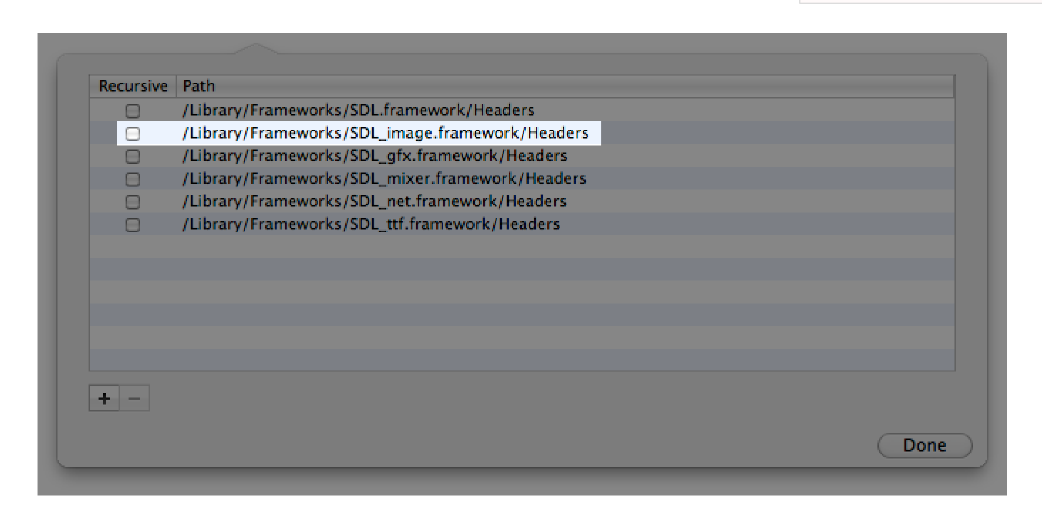
إذا كنت تعمل بالـVisual Studio فستكون العمليّة نفسها. لأنه إن تمكنت من تثبيت الـSDL لن يصعب عليك تثبيت الـSDL_Image.

تثبيت الـSDL_Image على Mac OS X

إن كنت تستعمل Mac OS X، نزّل الملف ذا الامتداد `.dmg` من موقع الـSDL و ضعه في المجلد `Library/Frameworks`.

اكتب بعد ذلك ”search paths“ في حقل البحث الخاص بـXcode. اعثر على السطر `Header search paths`، انقر مرتين على السطر من اليمين و أضف `/Library/Frameworks/SDL_image.framework/Headers`.

لم يبق لك سوى إضافة إطار العمل إلى المشروع. الصورة التالية توضح لك كيف يظهر الـ`Header search paths` بعد تثبيت الـSDL_image.



و بهذا يجب عليك تضمين الملف الرأسي في بداية الكود كالتالي :

```
#include "SDL_image.h"
```

في عوض استعمال الإشارتين `< >` قم باستعمال الكتابة السابقة فيما سأعطيك لاحقا.

تحميل الصور

الحقيقة أن تثبيت الـSDL_image أصعب بمئة مرة من استعمالها ! إنه عليك أنت تحديد صعوبة العمل بالمكتبة !

توجد دالة وحيدة عليك معرفتها : `IMG_Load`.
و هي تستقبل معاملا واحدا : اسم الملف الذي نريد فتحه.

و هذا أمر عملي لأن هذه الدالة تمكن من تحميل أي نوع من الملفات التي تتعامل معهاالـSDL_image (PNG، JPG، GIF و حتى الـTIF، إلخ). إذ تقوم وحدها بتحديد نوع الملف من خلال امتداده.

> ؟ بما أن الـSDL_Image تستطيع أيضاً فتح الصور BMP، فيمكنك الآن نسيان أمر استعمال الدالة `SDL_LoadBMP` و استعمال الدالة `IMG_Load` لتحميل كل أنواع الصور.

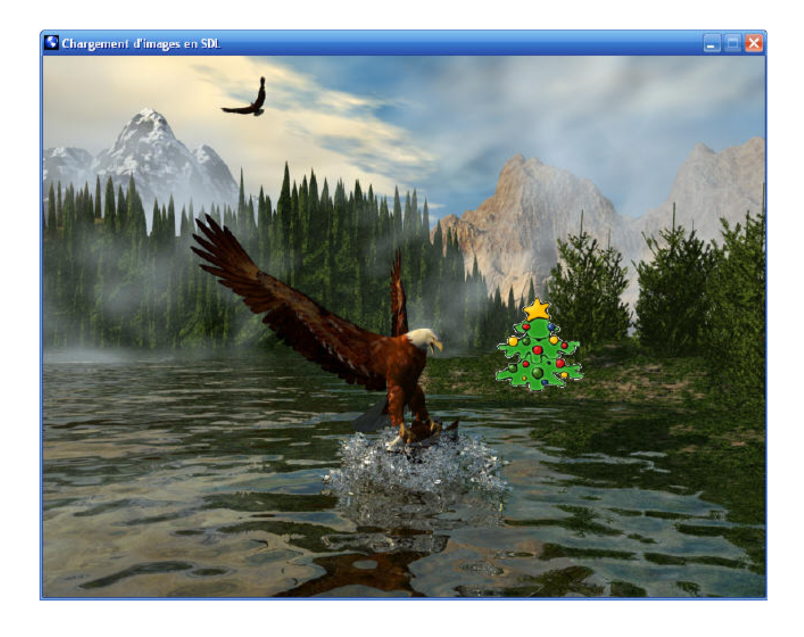
شيء جيد آخر : إذا كانت الصورة التي تحمّلها تملك الشفافية (كما هو حال الصور PNG و GIF) فإنّ SDL_Image تفعّل تلقائيّا الشفافية من أجل هذه الصورة ! مما يعني عدم وجود داعٍ لاستدعاء الدالة `SDL_SetColorKey`.

سأُقدّم لك الشفرة المصدرية التي تقوم بتحميل الصورة `sapin.png` و إظهارها.
لاحظ جيدا أنني قمت بتضمين `SDL/SDL_image.h` كما أنني لا استدعي الدالة `SDL_SetColorKey` لأن الصورة PNG التي استعملها شفافة طبيعيّا.
سترى أنني أستعمل الدالة `IMG_Load` في كلّ مكان بالشفرة و ذلك بتعويض الدالة `SDL_LoadBMP`.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <SDL/SDL.h>
4  /∗ including the header of SDL_image (adapt to your directory) ∗/
5  #include <SDL/SDL_image.h>
6  void pause();
7  int main(int argc, char ∗argv[])
8  {
9          SDL_Surface ∗screen = NULL, ∗backgoundImage = NULL, ∗sapin = NULL;
10         SDL_Rect backgoundPosition, sapinPosition;
11         backgoundPosition.x = 0;
12         backgoundPosition.y = 0;
13         sapinPosition.x = 500;
14         sapinPosition.y = 260;
15         SDL_Init(SDL_INIT_VIDEO);
16         SDL_WM_SetIcon(IMG_Load("sdl_icone.bmp"), NULL);
17         screen = SDL_SetVideoMode(800, 600, 32, SDL_HWSURFACE);
18         SDL_WM_SetCaption("Chargement d'images en SDL", NULL);
19         backgoundImage = IMG_Load("lac_en_montagne.bmp");
20         SDL_BlitSurface(backgoundImage, NULL, screen, &backgoundPosition);
21         /∗ Loading a PNG image with IMG_Load
22         We won't have any problem because the PNG image contains the
               transparency information inside ∗/
```

```
23          sapin = IMG_Load("sapin.png");
24          SDL_BlitSurface(sapin, NULL, screen, &sapinPosition);
25          SDL_Flip(screen);
26          pause();
27          SDL_FreeSurface(backgroundImage);
28          SDL_FreeSurface(sapin);
29          SDL_Quit();
30          return EXIT_SUCCESS;
31  }
32  void pause()
33  {
34          int cont = 1;
35          SDL_Event event;
36          while (cont)
37          {
38                  SDL_WaitEvent(&event);
39                  switch(event.type)
40                  {
41                          case SDL_QUIT:
42                          cont = 0;
43                  }
44          }
45  }
```

كما يمكننا الملاحظة، فقد تم دمج الصورة مع الخلفية بشكل ممتاز :



ملخّص

• تسمح الـSDL بتحميل صور على مساحات. افتراضيّاً، هي تسمح بالتعامل مع الصور ذات الصيغة BMP باستعمال الدالة `SDL_LoadBMP`.

- يمكننا تعريف لون شفاف باستعمال الدالة `SDL_SetColorKey` .
- يمكننا جعل الصورة أكثر أو أقل شفافية و ذلك باستعمال الدالة `SDL_SetAlpha` .
- المكتبة SDL_image تسمح بإدخال صور من أيّة صيغة كانت (JPG، PNG، ...) باستعمال الدالة `IMG_Load` . لكن علينا تسطيب هذه المكتبة بالإضافة إلى الـSDL.

الفصل 23

معالجة الأحداث (Event handling)

معالجة الأحداث هو من أهم الأساسيات في الـSDL.
و ربّما قد يكون الشطر الأكثر شغفاً لاكتشافه. لأنه انطلاقا من هنا ستبدأ فعلاً في التحكّم في تطبيقك.

كلّ من مرفقات الحاسوب (فأرة، لوحة مفاتيح، ...) قادرة على إنتاج حدث. سنتعلّم كيف نستقبل كل حدث و نتعامل معه. تطبيقك سيصبح أخيراً تفاعليّا !

فعلياً، ما هو الحدث ؟ الحدث هو عبارة عن إشارة (signal) يتم إرسالها عن طريق إحدى مرفقات الحاسوب (peripherals) (أو عن طريق نظام التشغيل بذاته) إلى التطبيق. هذه أمثلة عن بعض الأحداث المألوفة :

- حينما يضغط المُستعمل على زر من لوحة المفاتيح.
- و أيضاً حينما ينقر بالفأرة.
- حينما يحرّك الفأرة.
- حينما يقوم بتصغير النافذة.
- حينما يطلب إغلاق النافذة.
- إلى آخره.

الهدف من هذا الفصل هو تعلّم كيفية معالجة الأحداث. يمكنك أخيراً القول للحاسوب : "إذا نقر المستعمل في هذا المكان، قم بفعل كذا، و إن لم يفعل، قم بكذا. إذا حرّك الفأرة، قم بكذا. إذا ضغط على الزر Q، أوقف البرنامج. إلخ".

1.23 مبدأ عمل الأحداث

لنتعوّد على الأحداث، سنتعلّم كيف نتعامل مع أسهل حدث : **طلب غلق البرنامج**. هذا حدث يُنتجُ حينما يقوم المستعمل بالنقر على الزر X :



إنه فعلاً الحدث الأكثر سهولة. إضافة على ذلك، هو حدث قد استعملته سابقاً دون أن تعلم بذلك لأنه متواجد في الدالة `pause` !
بالفعل، دور هذه الدالة هو انتظار المستعمل حتّى يقرّر غلق البرنامج، لأننا لو لم نستعملها كانت النافذة لتظهر و تختفي بسرعة البرق !

> م يمكنك من الآن نسيان الدالة `pause` . قم بحذفها من الشفرة المصدرية لأننا سنتعلّم كيف نكتب محتواها بأنفسنا.

متغيّر الحدث

لمعالجة الأحداث، ستحتاج إلى التصريح عن متغيّر (واحد فقط، كن متأكدا) من نوع `SDL_Event`.
فلتقم بتسميته بالاسم الذي يحلو لك، أنا سأسمّيه `event` ، و هي تعني "حدث" بالإنجليزيّة.

```
SDL_Event event;
```

من أجل اختبارات الشفرة، سنستعمل دالة `main` بسيطة للغاية تقوم بإظهار نافذة فقط، مثلما رأينا في الفصول الأولى. هذا ما يجب أن تبدو عليه الدالة `main` :

```
int main(int argc, char *argv[])
{
        SDL_Surface *screen = NULL;
        SDL_Event event; // This variable will help us to manage the events
        SDL_Init(SDL_INIT_VIDEO);
        screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
        SDL_WM_SetCaption("Gestion des événements en SDL", NULL);
        SDL_Quit();
        return EXIT_SUCCESS;
}
```

إذا، هي شفرة بدائية جداً و هي لا تحوي سوى شيء جديد : تعريف المتغير `event` الذي سنستعين به قريباً.

جرّب الشفرة : مثلما توقّعنا، يجدر بالنافذة أن تظهر و تختفي في لحظة.

حلقة الأحداث

حينما نريد انتظار حدث، نستعمل غالباً حلقة. هذه الحلقة التكرارية تستمر في الاشتغال مادُمنا لم نستقبل الحدث المُراد.
يجب علينا أن نستعمل متغيراً منطقياً لكي يحدد لنا ما إن كان علينا البقاء في الحلقة أو الخروج منها.
أنشئ هذا المتغير و سمّه مثلا `cont` [1] :

[1] إذا كنت تفكّر في تسمية `continue` فلا تفعل، لأنّها كلمة مفتاحيّة (محجوزة)، و بالتالي لا يمكن استخدامها كاسم لمتغير.

```
int cont = 1;
```

هذا المتغير المنطقي يأخذ القيمة 1 في البداية لأننا نريد للحلقة أن تتكرر مادام المتغير `cont` يحمل هذه القيمة (صحيح). ما إن يأخذ المتغير المنطقي القيمة 0 (خطأ)، نخرج من الحلقة و يتوقف البرنامج.

هذا ما تبدو عليه الحلقة :

```
while (cont)
{
        // Dealing with the event
}
```

هكذا إذاً : لدينا لحدّ الآن حلقة غير منتهية لا تنتهي إلا إذا أخذ المتغير `cont` القيمة 0. الأكثر أهمية هو ما نكتبه في داخل تلك الحلقة.

استرجاع الحدث

الآن سنقوم باستدعاء دالة من الـSDL لكي تحقق ما إن تم إنتاج حدث.
لدينا دالتان للقيام بهذا العمل، لكن كلا منهُما تعمل بطريقة مختلفة عن الأخرى :

- `SDL_WaitEvent` : تقوم بانتظار إنتاج حدث. هذه الدالة نقول عنها تعطيلية لأنها توقف عمل البرنامج مادام لم يتم إنتاج أي حدث.
- `SDL_PollEvent` : هذه الدالة تقوم بنفس العمل لكنها ليست تعطيلية. لأنها تُخبرنا ما إن تم إنتاج حدث أم لا، فإن لم يكن هناك أي حدث فإنها تعيد التحكّم إلى البرنامج مباشرة.

هاتان الدالّتان مهمّتان، لكن في حالتين مختلفتين.
لتبسيط الأمور، إذا استعملت `SDL_WaitEvent` فإن برنامجك لن يتُعب كثيراً المُعالج لأنه سيتوقف مُنتظراً إنتاج حدث.
بالمُقابل، إذا استعملت `SDL_PollEvent`، سيقوم البرنامج بالعمل على الحلقة `while` و استدعاء الدالة `SDL_PollEvent` بشكل غير معرّف إلى حين إنتاج حدث معين. و بهذا تستعمل المُعالج بنسبة 100 %.

> ؟ لكن ألا يجب أن نستعمل دائماً الدالة `SDL_WaitEvent` بما أنها لا تستعمل المُعالج كثيراً ؟

كلّا، لأنه توجد حالات لا يمكن الاستغناء فيها عن الدالة `SDL_PollEvent`. و هي حالة الألعاب التي يتم فيها تحديث الشاشة حتى و إن لم يكن هناك أي حدث.
فلنأخذ مثلاً اللعبة Tetris : تقوم الكتل بالنزول لوحدها، لا يحتاج المُستعمل إلى إنتاج حدث من أجل حصول هذا الأمر ! لو استعملنا `SDL_WaitEvent`، سيبقى البرنامج معُطّلا و لن تتمكّن من تحديث الشاشة لإنزال الكتل !

؟

ماذا تفعل `SDL_WaitEvent` لكي لا تستهلك من المُعالج كثيراً ؟
فبعد كل شيء، الدالة مُجبرة على البقاء في حلقة غير منتهية لكي تختبر كلّ الوقت ما إن كان هناك حدث أم لا، أليس كذلك ؟

الحقيقة أنني كنت أطرح هذا السؤال قبل وقت قليل. الإجابة معقّدة قليلاً لأنها تخصّ الطريقة التي يتحكّم فيها النظام بالعمليّات (Processes) (البرامج التي هي في طور الاشتغال).
إذا كنت تريد -لكنّي سأتحدّث بسرعة-، بالنسبة للدالة `SDL_WaitEvent`، عمليّة البرنامج تُوضع في طور الانتظار. إذا فإن البرنامج لا يعمل عليه المعالج بعد تلك اللحظة.
سيتم "إيقاظه" من طرف نظام التشغيل حينما يتم إنتاج حدث. يعني أن المعالج سيعود إلى العمل على البرنامج في هذه اللحظة. هذا ما يشرح لِمَا لا يستهلك البرنامج من المعالج شيئا بينما يكون في طور انتظار الحدث.

أدري أن هذه المفاهيم تبدو مجرّدة لك الآن. لكنك لست مُجبراً على فهم كل هذا الآن لأنك ستبدأ في التأقلم مع هذه المعلومات شيئاً في شيئاً مع التطبيق.
الآن سنستعمل `SDL_WaitEvent` لأن البرنامج سيبقى بسيطا باستخدامها. على أي حال فالتعامل مع هاتين الدالتين لن يتغير من واحدة إلى أخرى.

يجب أن تبعث للدالة عنوان المتغير `event` الذي يقوم بتخزين الحدث.
بما أن هذا المتغير ليس عبارة عن مؤشّر (أعد رؤية طريقة التصريح به أعلاه)، سنستعمل الاشارة `&` قبل اسم المتغير و ذلك لنُعطي عنوانه :

```
SDL_WaitEvent(&event);
```

بعد استدعاء هذه الدالة، المتغير `event` يحتوي إجبارياً حدثاً ما.

م

هذه الحالة ليست نفسها لو استعملنا `SDL_PollEvent` لأن هذه الأخيرة قادرة على أن تُرجع لنا : "لا يوجد أي حدث".

تحليل الحدث

الآن نحن نتوفر على متغير `event` يحتوي على معلومات حول الحدث الذي تم إنتاجه.
يجب أن نرى المركّب `event.type` و نختبر قيمته. غالبا ما نستعمل `switch` لاختبار الحدث.

؟

لكن كيف لنا أن نعرف ما هي القيمة الموافقة للحدث "أغلق البرنامج" مثلا ؟

الـSDL توفّر لنا بعض الثوابت، مما يسهّل كثيراً كتابة البرنامج. هذه الثوابت كثيرة العدد (بقدر وجود أحداث ممكن حصولها في الحاسوب). سنتعرّف على هذه الثوابت بتقدّمنا في هذا الفصل.

```
while (cont)
{
        SDL_WaitEvent(&event); // Getting the event in "event"
        switch(event.type) // Testing the event's type
        {
                case SDL_QUIT: // If it's a quit event
                cont = 0;
                break;
        }
}
```

هكذا تعمل الشفرة :

1. ما إن يتم انتاج حدث، تُرجع الدالة `SDL_WaitEvent` الحدث في المتغير `event`.
2. نقوم بتحليل نوع الحدث بالاستعانة بـ `switch` . نوع الحدث موجود في `event.type` .
3. نختبر بمساعدة `case` نوع الحدث. لحدّ الآن، نحن لا نحقق إلا إذا ما كان الحدث يوافق `SDL_QUIT` (طلب إغلاق البرنامج)، لأنّها الحالة الوحيدة التي تهمّنا.
4. إذا كان الحدث هو `SDL_QUIT` ، فهذا يعني أن المستعمل طلب إغلاق البرنامج. في هذه الحالة، نعطي للمتغير المنطقي `cont` القيمة 0. في الدورة القادمة للحلقة، سيكون الشرط غير محقق، فيتوقف تشغيل البرنامج.
5. إذا لم يكن الحدث هو `SDL_QUIT` ، مما يعني أنه قد حدث شيء آخر : قام المستعمل بالضغط على زر، بالنقر على الفأرة أو ببساطة قام بتحريك الفأرة داخل النافذة. و بما أن هذه الأحداث لا تهمّنا، لن نقوم بمعالجتها. لن نقوم إذا بأي شيء : تقوم الحلقة بالانتقال في كلّ مرة إلى دورة جديدة ننتظر فيها وقوع حدث جديد (بمعنى آخر، نعود إلى النقطة 1).

ما أنا أشرحه لك الآن هو أمر مهم جداً. إذا فهمت هذه الشفرة، فقد فهمت كلّ شيء و سيكون باقي الفصل سهلاً للغاية.

الشفرة الكاملة

```
int main(int argc, char *argv[])
{
        SDL_Surface *screen = NULL;
        SDL_Event event; // The event's variable
        int cont = 1; // A boolean for the loop
        SDL_Init(SDL_INIT_VIDEO);
        screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
        SDL_WM_SetCaption("Gestion des événements en SDL", NULL);
        while (cont) // While the variable's value is
        {           // not equal to 0
                SDL_WaitEvent(&event); // We wait for an event that we
                    recuperate in "event"
```

```
12                  switch(event.type) // Testing the event's type
13                  {
14                          case SDL_QUIT: // If it's a quit event
15                          cont = 0; // We change the boolean value so we go out
                                from the loop.
16                          break;
17                  }
18          }
19          SDL_Quit();
20          return EXIT_SUCCESS;
21  }
```

هاهي الشفرة الكاملة. لا يوجد شيء صعب : إذا قمت بمتابعة الفصل إلى الآن، يجدر بك أن تكون قد فهمت كلّ شيء. على أي حال فقد لاحظت أنّنا لم نقم إلا بإعادة كتابة ما تقوم به الدالة `pause` . قارن هذه الشفرة بما تقوم به الدالة `pause` : هو نفس الشيء، إلا أنه في هذه الحالة نقوم بوضع كلّ شيء في الدالة `main` . بالطبع، من المستحسن نقل الشفرة إلى دالة أخرى على حدى كـ`pause` ، لأن ذلك سيقلّل من حجم الدالة `main` و يجعلها أفضل من ناحية فهم الشفرة.

2.23 لوحة المفاتيح

سنقوم الآن بدراسة الأحداث التي تُنتج عن طريق لوحة المفاتيح.

إذا فهمت بداية الفصل، فلن تواجه أي مشكل في التعامل مع أي نوع من الأحداث. لا يوجد ماهو أسهل.

لماذا هذا سهل ؟ لأننا الآن فهمنا طريقة عمل الحلقة التكرارية غير المنتهية، كل ما ستقوم بفعله هو إضافة بعض الحالات إلى الـ`switch` من أجل تحليل أنواع أخرى من الأحداث. لا يفترض أن يكون هذا الأمر صعباً.

أحداث لوحة المفاتيح

يوجد نوعان من الأحداث التي يمكن توليدها عن طريق لوحة المفاتيح :

- `SDL_KEYDOWN` : حينما يتم بدأ الضغط على زر من لوحة المفاتيح.
- `SDL_KEYUP` : حينما يحرّر زر لوحة المفاتيح.

لماذا يوجد حدثان إثنان ؟
لأننا حينما نضغط على زر، يحدث أمران : شدّ الزر إلى الأسفل (`SDL_KEYDOWN`) ثمّ تحريره (`SDL_KEYUP`). تسمح لنا الـSDL بتحليل كل من هذين الحدثين على حدى، و هذا أمر عملي جداً، سترى ذلك.

لحدّ الآن سنكتفي بتحليل الحدث `SDL_KEYDOWN` (الضغط على الزر) :

```
while (cont)
{
        SDL_WaitEvent(&event);
        switch(event.type)
        {
                case SDL_QUIT:
                cont = 0;
                break;
                case SDL_KEYDOWN: // If we press a button
                cont = 0;
                break;
        }
}
```

إذا ضغطنا على أي زر سيتوقف البرنامج، جرّب ذلك !

استرجاع رمز الزر

معرفة أنه تم الضغط على زر من لوحة المفاتيح هو أمر جيد، لكن معرفة أي الأزرار تم الضغط عليه بالضبط هو أمر أحسن !

يمكننا معرفة الزر الذي تم الضغط عليه بفضل مركّب مركّب مركّب المتغيّر (أوف !) و الذي يُدعى `event.key.keysym.sym`. هذا المتغير يحتوي قيمة الزر الذي تم الضغط عليه (و هو يعمل حتى في الحين الذي نحرر فيه الزر `SDL_KEYUP`).

> الشيء الجيد هو أن الـSDL تسمح باسترجاع هذه القيمة من كل أزرار لوحة المفاتيح و التي تتضمّن على الحروف و الأرقام، و كذلك الأزرار `Esc`، `Print scr.`، `Del`، `Enter`، ... إلخ.

يوجد ثابت من أجل كل زر في اللوحة. يمكنك الاطلاع على قائمة هذه الثوابت من خلال الملفات التوثيقيّة الخاصة بالـSDL، الّتي من المفترض أنّك قد نزلتها مع المكتبة SDL.
إن لم تفعل، فأنصحك بالتوجه إلى موقع المكتبة و تحميل هذه الملفات لأنها مهمّة للغاية.

ستجد قائمة أزرار لوحة المفاتيح في القسم "Keysym definitions". هذه القائمة طويلة جداً و لا يمكنني تقديمها هنا و لهذا عليك تصفح التوثيق من الموقع مباشرة.

http://www.siteduzero.com/uploads/fr/ftp/mateo21/sdlkeysym.html

هذه الملفات مُحررة باللغة الانجليزية، و هي غير متوفرة بلغة أخرى. إذا كنت تريد البرمجة حقّاً، فمن الواجب أن تجيد هذه اللغة لأنّ كلّ الملفات التوثيقيّة مكتوبة بها، فلا يمكنك أبداً تجاوزها !

يوجد في اللائحة جدولان : واحد كبير (في البداية) و آخر صغير (في النهاية). نحن الآن نهتم بالجدول الأكبر.
في العمود الأول تجد الثابت، في العمود الثاني تجد القيمة الموافقة له بالـASCII و أخيراً في العمود الثالث تجد وصفاً للزر.
لاحظ أن بعض الأزرار كـ`Maj` (`Shift`) لا تملك قيمة ASCII موافقة لها.

فلنأخذ مثلاً الزر `Esc`. يمكننا معرفة ما إن كان هذا الزر مضغوطاً كالتالي :

```
switch (event.key.keysym.sym)
{
        case SDLK_ESCAPE: // Pressing Escape button lets us quit the program
        cont = 0;
        break;
}
```

أستعمل `switch` من أجل الاختبار الأول لكن كان بإمكاني استعمال `if` ببساطة.
في كلّ مرة أميل إلى الاستعانة بالـ`switch` حينما أعالج الأحداث لأنني أختبر الكثير من القيم المختلفة (عملياً، يتوفر لدينا الكثير من الحالات في الـ`switch`، على عكس هذا المثال).

هذه حلقة حدث كاملة يمكنك تجريبها :

```
while (cont)
{
        SDL_WaitEvent(&event);
        switch(event.type)
        {
                case SDL_QUIT:
                cont = 0;
                break;
                case SDL_KEYDOWN:
                switch (event.key.keysym.sym)
                {
                        case SDLK_ESCAPE:
                        cont = 0;
                        break;
                }
                break;
        }
}
```

هذه المرة، يتوقف البرنامج حينما نضغط على الزر `Esc` أو إذا نقرنا على الرمز `X` أعلى النافذة. و الآن بما أنك تعرف كيف تغلق البرنامج بالضغط على زر معين، أنت مخوّل لاستعمال وضع الشاشة الكاملة إذا كان هذا ممتعا لك (استعمل العلَم `SDL_FULLSCREEN` في الـ`SDL_SetVideoMode`، كتذكير).
سابقاً كنت قد منعتك من استعمال هذا الأسلوب في العرض خشية أننا لن تمكن من غلق البرنامج (لأنه لن يظهر لنا زرّ الإغلاق الّذي نضغط عليه لإيقاف البرنامج !)

3.23 تمرين : تحريك Zozor بواسطة لوحة المفاتيح

أنت الآن قادر على تحريك صورة في النافذة بواسطة لوحة المفاتيح !

هذا تمرين مهمّ جداً سيسمح لنا بالتعرف على كيفية استعمال الـdouble buffering و الاستعمال المتكرر للأزرار.
إضافة إلى ذلك، ما أنا بصدد تعليمه لك هو قاعدة كل ألعاب الفيديو التي تُصنع بالـSDL. و لهذا فإن **هذا التمرين ليس اختياريا** ! أدعوك لقراءته و محاولة حلّه بشكل جدّي.

تحميل الصورة

في البداية، سنقوم بتحميل صورة. سيكون الأمر بسيطاً : سنعيد استعمال صورة Zozor المُستعملة في الفصل السابق.
أنشئ المساحة `zozor`، حمّل الصورة و حوّل خلفيتها إلى اللون الشفاف (أذكرك بأن صيغة الصورة هي BMP).

```
zozor = SDL_LoadBMP("zozor.bmp");
SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));
```

بعد ذلك، الشيء الأكثر أهمية، يجب عليك إنشاء متغير من نوع `SDL_Rect` لتقوم بحِفظ مركبات Zozor.

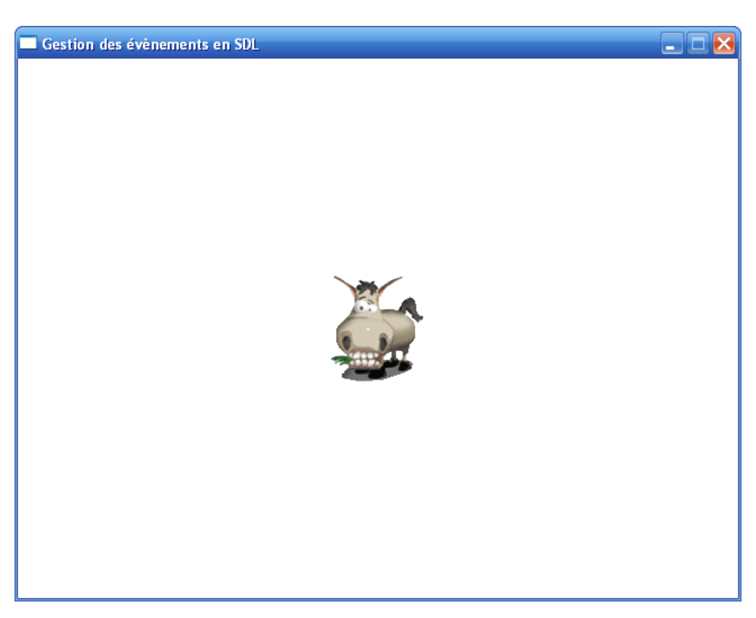
```
SDL_Rect positionZozor;
```

أنصحك بإعطاء قيم ابتدائية للمركبات، ضع مثلا `x = 0` و `y = 0` (الوضعية أعلى يسار النافذة) أو قم بِمركَزَة Zozor في وسط النافذة كما قمت بتعليمك هذا من قبل.

```
// We center Zozor in the screen
zozorPosition.x = screen->w / 2 - zozor->w / 2;
zozorPosition.y = screen->h / 2 - zozor->h / 2;
```

> X
> يجب عليك تهيئة المتغير `zozorPosition` بعد تحميل المساحتين `screen` و `zozor`. في الواقع، سأستعمل العرض `w` و الارتفاع `h` لهاتين المساحتين من أجل حساب الموقع المركزي لـZozor في الشاشة، و لهذا كان لازما أن يتم تهيئة هاتين المساحتين من قبل.

إذا كنت قد تدبّرت أمرك جيّداً، يجدر أن يظهر Zozor في وسط النافذة.



لقد اخترت هذه المرة وضع خلفية بيضاء (قمت بـ `SDL_FillRect`) لكن هذا ليس واجباً.

مخطط البرمجة بالأحداث

حينما تقوم ببرمجة برنامج يتفاعل مع الأحداث (كما سنقوم بفعله الآن)، يجب عليك اتّباع نفس "المخطط" في غالب الأحيان.
يجدر بك حفظ هذا المخطط عن ظهر قلب :

```
while (cont)
{
        SDL_WaitEvent(&event);
        switch(event.type)
        {
                // Managing the events of type SOMETHING
                case SDL_SOMETHING:
                // Managing the events of type ANOTHERTHING
                case SDL_ANOTHERTHING:
        }
        // We clear the screen
        SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 255, 255, 255));
        // We do all the necessary SDL_BlitSurface to past the surfaces on the
            screen
        // We update the display
        SDL_Flip(screen);
}
```

سأقدّم لك أهمّ السطور التي تكوّن الحلقة الرئيسية في برنامج SDL.
سنستمر في تشغيل الحلقة مادام لم يتم طلب غلق البرنامج.

1. ننتظر حدثاً (`SDL_WaitEvent`) أو نقوم بالتحقق من وجود حدث لكن لا نقوم بانتظار حدوث واحد (`SDL_PollEvent`). حاليا نكتفي باستعمال `SDL_WaitEvent` .
2. نستعمل `switch` (كبير) من أجل معرفة نوع الحدث الذي نحن نتعامل معه. نقوم بتحليل الحدث الذي تلقّيناه ثم نقوم ببعض الحسابات و العمليات.
3. ما إن نخرج من الـ `switch` ، نحضّر عرضا جديدا لنقوم بإظهاره.
4. أول شيء نفعله : نمسح الشاشة باستعمال `SDL_FillRect` . إن لم نقم بذلك، ستبقى بعض "آثار" الشاشة السابقة في الشاشة الحالية مما يشوش المظهر.
5. نقوم بعد ذلك بلصق كل المساحات على الشاشة.
6. أخيرا، ما إن ننتهي من كل ذلك، نقوم بتحديث العرض من أجل المستعمل و ذلك باستدعاء الدالة `SDL_Flip` .

معالجة الحدث SDL_KEYDOWN

لنرى الآن كيف نعالج الحدث `SDL_KEYDOWN`.
هدفنا هو تحريك Zozor بواسطة لوحة المفاتيح باستعمال الأسهم التوجيهية. و لهذا سنقوم بتغيير مركّباته في الشاشة بدلالة السهم الذي يضغط عليه المستعمل :

```
switch(event.type)
{
        case SDL_QUIT:
        cont = 0;
        break;
        case SDL_KEYDOWN:
        switch(event.key.keysym.sym)
        {
                case SDLK_UP: // Up arrow
                zozorPosition.y--;
                break;
                case SDLK_DOWN: // Down arrow
                zozorPosition.y++;
                break;
                case SDLK_RIGHT: // Right arrow
                zozorPosition.x++;
                break;
                case SDLK_LEFT: // Left arrow
                zozorPosition.x--;
                break;
        }
        break;
}
```

أين وجدت أسماء الثوابت ؟ لقد وجدتها في التوثيق !
لقد اعطيتك قبل قليل الرابط الخاص بالتوثيق الّذي يعطي لائحة كل الأزرارفي لوحة المفاتيح : هنا وجدت ضالّتي.

ما فعلناه هنا هو أمر بسيط جداً :

- إذا ضغطنا على السهم "أعلى"، نقوم بإنقاص الترتيبة (`y`) الخاصة بـZozor ببيكسل واحد من أجل جعله يصعد. لاحظ أننا لسنا مُجبرين على تحريكه ببيكسل واحد، يمكننا تحريكه بـ10 بيكسل في كلّ مرة.
- إذا توجهنا إلى الأسفل، سنقوم بزيادة الترتيبة (`y`) لـZozor.
- إذا توجهنا لليمين نزيد قيمة الفاصلة (`x`).
- إذا توجهنا لليسار نقوم بإنقاص الفاصلة (`x`).

و الآن ؟
بما أنني أعطيتك التوجيهات و حتى المخطط، يجدر بك أن تكون قادراً على كتّابة الشفرة التي تسمح بتحريك Zozor في النافذة !

```c
int main(int argc, char *argv[])
{
	SDL_Surface *screen = NULL, *zozor = NULL;
	SDL_Rect zozorPosition;
	SDL_Event event;
	int cont = 1;
	SDL_Init(SDL_INIT_VIDEO);
	screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
	SDL_WM_SetCaption("Gestion des événements en SDL", NULL);
	// Loading Zozor
	zozor = SDL_LoadBMP("zozor.bmp");
	SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0,
		255));
	zozorPosition.x = screen->w / 2 - zozor->w / 2;
	zozorPosition.y = screen->h / 2 - zozor->h / 2;
	while (cont)
	{
		SDL_WaitEvent(&event);
		switch(event.type)
		{
			case SDL_QUIT:
			cont = 0;
			break;
			case SDL_KEYDOWN:
			switch(event.key.keysym.sym)
			{
				case SDLK_UP: // Up arrow
				zozorPosition.y--;
				break;
				case SDLK_DOWN: // down arrow
				zozorPosition.y++;
				break;
				case SDLK_RIGHT: // Right arrow
				zozorPosition.x++;
				break;
				case SDLK_LEFT: // Left arrow
				zozorPosition.x--;
				break;
			}
			break;
		}
		// Clear the screen
		SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 255, 255,
			255));
		// We put zozor in its new position
		SDL_BlitSurface(zozor, NULL, screen, &zozorPosition);
		// Update the display
		SDL_Flip(screen);
	}
```

```
48          SDL_FreeSurface(zozor);
49          SDL_Quit();
50          return EXIT_SUCCESS;
51  }
```

إنه من الضروري جداً الفهم الجيد لكيفية عمل الحلقة الرئيسية في البرنامج. يجب أن تكون قادراً على كتابتها بنفسك. استعن بالمخطط الذي أعطيتك إياه أعلاه إذا اقتضت الحاجة.

باختصار إذا، توجد حلقة كبيرة تُسمى "الحلقة الرئيسية في البرنامج". و هي لا تتوقف إلا إذا أعطينا للمتغير `cont` القيمة 0.

في هذه الحلقة، نقوم أولا باسترجاع حدث لنقوم بمعالجته. نستعمل `switch` من أجل تحديد نوع الحدث. بدلالة الحدث، نقوم بعمليات مختلفة. في حالتنا هذه، نقوم بتحديث مركبات وضعية Zozor من أجل إعطاء انطباع أننا نقوم بتحريكه.

ثم بعد الـ`switch` يجب عليك تحديث الشاشة كالتالي :

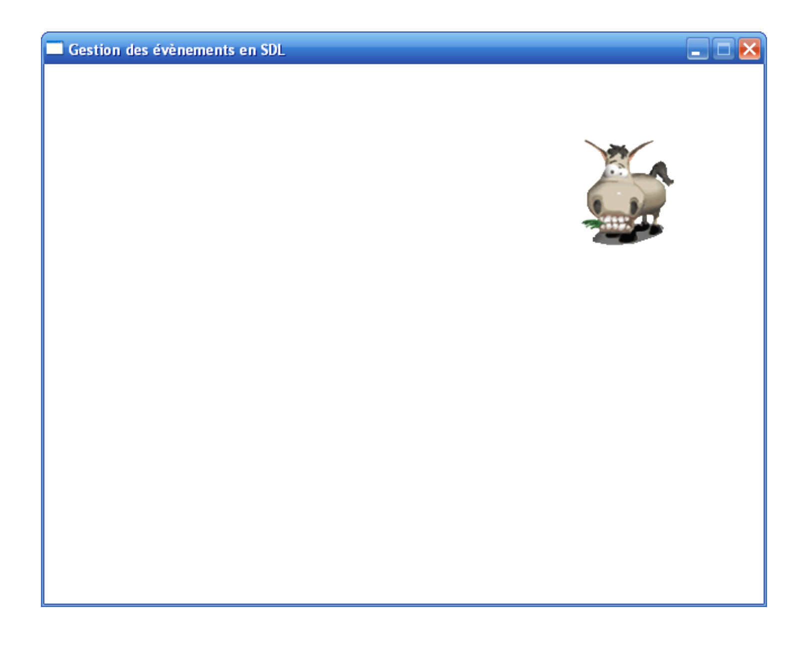
1. أولاً، نمسح الشاشة باستعمال `SDL_FillRect` (باستعمال لون خلفية يناسبك).

2. ثم تقوم بتسوية المساحات على الشاشة. هنا، لم أحتج إلى لصق إلا Zozor لأننا لا نحتاجه إلا هو كما هو واضح، و من المهم جداً أن نضع Zozor في الموضع `zozorPosition` ! لان هذا ما يصنع الفارق : إذا كنتُ قد حدّثتُ `zozorPosition` كان Zozor ليظهر في مكان آخر و بهذا نعتقد أننا غيّرنا مكانه كليا !

3. أخيراً، و آخر شيء للقيام به : `SDL_Flip` لكي نحدّث الشاشة من أجل المستعمل.



و بهذا نتمكن من تحريك الحيوان إلى أي مكان نريده !



بعض التحسينات

تكرار الضغط على الأزرار

لحد الآن، البرنامج يعمل لكنّه يُلزِم تحرّك اللاعب أن يكون بيكسلا في المرّة الواحدة. نحن مُجبَرون على الضغط من جديد على الأسهم إذا أردنا التحرك مرة أخرى ببيكسل. لا أدري بالنسبة لك، لكنني أستمتع أحيانا بالبقاء ضاغطا على نفس الزر وقتا أطول لكي أحرك اللاعب بـ200 بيكسل.

على كل حال، من حسن الحظّ أن الدالة `SDL_EnableKeyRepeat` موجودة !
هذه الدالة تسمح بتفعيل الضغط المتكرر على الأزرار. فهي تحرّض الـSDL على إعادة إنتاج حدث من نوع `SDL_KEYDOWN` إذا بقي المستعمل ضاغطاً على نفس الزر لمدة من الزمن.

يمكنك استدعاء هذه الدالة أينما أردت، لكنني أنصحك باستدعائها قبل الحلقة الرئيسية للبرنامج. يمكن للدالة أخذ معاملين :

- المدة (بالميلّي ثانية) التي يجدر بالزر أن يبقى فيها مضغوطاً قبل تفعيل تكرار الضغط على الأزرار.
- الأجل (بالميلّي ثانية) بين كل إنتاج لحدث `SDL_KEYDOWN` و آخر ما إن يتم تفعيل تكرار الضغط على الأزرار.

المعامل الأول يشير إلى مدة الزمن التي يجدر بنا بعدها إنتاج تكرار الضغط على الأزرار في المرة الأولى. أما الثاني فيشير إلى الوقت اللازم ليتم إعادة إنتاج الحدث.
شخصياً، و من أجل أسباب ليونة التحرك، أعطي غالباً نفس القيمة للمعاملين. جرّب القيمة 10 ميلّي ثانية :

```
SDL_EnableKeyRepeat(10, 10);
```

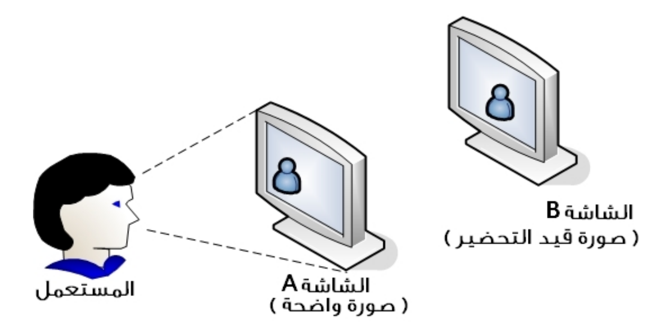
الآن، يمكنك البقاء ضاغطاً على نفس السهم. سترى أنّ هذا أحسن !

العمل بالـdouble buffering

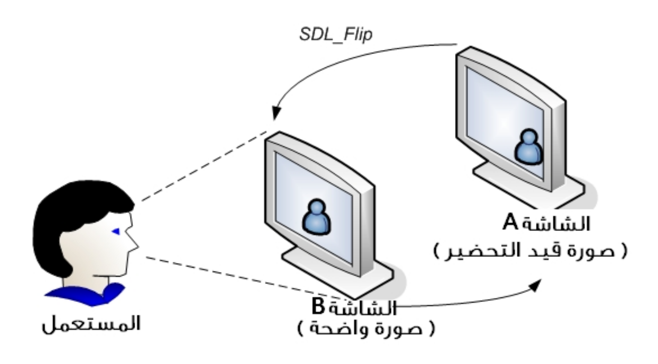
انطلاقا من الآن، سيكون من الجيد تفعيل تقنية الـdouble buffering الخاصة بالـSDL. الـdouble buffering هي تقنية مستعملة بكثرة في الألعاب. تسمح هذه التقنيّة بتجنب التقطع في الصورة.
لِمَا تتقطع الصورة ؟ لأنه حينما نرسم في الشاشة، المستعمل "يرى" كيف ترسم و بهذا يرى كيف تُمسح الشاشة. حتى وإن جرت العملية بسرعة فإن مخّ الإنسان يلتقط إشارات خفيفة و قد تكون مزعجة.

تقنية الـdouble buffering تعمل على استخدام "شاشتين" : واحدة حقيقية (التي يراها المستعمل في شاشة الحاسوب) و أخرى افتراضية (هي صورة يقوم الحاسوب بإنشائها في الذاكرة).

هاتان الشاشتان تتناوبان : الشاشة A تظهر في حين تحضّر الشاشة B الصورة القادمة في الخلفية. لاحظ الصورة التالية :



ما إن يتم رسم الصورة في الشاشة الخلفية (الشاشة B)، نقوم بقلب الشاشتين و ذلك باستدعاء الدالة `SDL_Flip`.



الشاشة A تصبح شاشة خلفية و تقوم بتحضير الصورة القادمة، بينما يتم إظهار الصورة في الشاشة B و يراها المستعمل.
النتيجة : لا يوجد تقطع في الصورة !

لتحقيق هذا كل ما عليك فعله هو تحميل وضع العرض بإضافة العَلَم `SDL_DOUBLEBUF` :

```
screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
```

لا يوجد شيء آخر لتغييره في الشفرة المصدرية.

> الـdouble buffering هي تقنية معروفة جداً في البطاقة الرسومية (Graphics card) للحاسوب. ما أقصده هو أن الجهاز هو من يتحكم في كلّ شيء، و يتم ذلك بسرعة جدّ فائقة.

ستتساءل ربّما لماذا قد كنا استعملنا `SDL_Flip` من قبل دون الـdouble buffering ؟
الواقع أن لهذه الدالة وظيفتين :

- إذا كانت الـdouble buffering مفعّلة، فستتحكم في تناوب الشاشتين.
- أما إن كانت غير مفعّلة، فهي تتحكم في تحديث النافذة يدويا. هذه التقنية تعمل في حالة كان البرنامج لا يقدّم حركية كبيرة، و لكن في غالبية الألعاب، أنصحك بتفعيلها.

من الآن و صاعداً، سأقوم بتفعيل هذه التقنية في كل الشفرات المصدرية التي أكتبها (لأنها لا تكلف الكثير و تقدم الكثير، فما نشكي ؟)

إليك الشفرة المصدرية الكاملة التي تسمح باستعمال الـdouble buffering و تكرار الضغط على الأزرار. إنها مشابهة للشفرة التي رأيناها قبل قليل، لقد قمت فقط بإضافة بعض التعليمات التي نحن بصدد تعلّمها :

```c
int main(int argc, char *argv[])
{
        SDL_Surface *screen = NULL, *zozor = NULL;
        SDL_Rect zozorPosition;
        SDL_Event event;
        int cont = 1;
        SDL_Init(SDL_INIT_VIDEO);
        zozorPosition = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE |
            SDL_DOUBLEBUF); // Double buffering
        SDL_WM_SetCaption("Gestion des événements en SDL", NULL);
        zozor = SDL_LoadBMP("zozor.bmp");
        SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0,
             255));
        zozorPosition.x = screen->w / 2 - zozor->w / 2;
        zozorPosition.y = screen->h / 2 - zozor->h / 2;
        SDL_EnableKeyRepeat(10, 10); // Enabling keys repetition
        while (cont)
        {
                SDL_WaitEvent(&event);
                switch(event.type)
                {
                        case SDL_QUIT:
                        cont = 0;
                        break;
                        case SDL_KEYDOWN:
                        switch(event.key.keysym.sym)
                        {
                                case SDLK_UP:
                                zozorPosition.y--;
                                break;
                                case SDLK_DOWN:
                                zozorPosition.y++;
                                break;
                                case SDLK_RIGHT:
                                zozorPosition.x++;
                                break;
                                case SDLK_LEFT:
                                zozorPosition.x--;
                                break;
                        }
                        break;
                }
                SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 255, 255,
                     255));
```

```
42              SDL_BlitSurface(zozor, NULL, screen, &zozorPosition);
43              SDL_Flip(screen);
44          }
45          SDL_FreeSurface(zozor);
46          SDL_Quit();
47          return EXIT_SUCCESS;
48  }
```

4.23 الفأرة

ربما تعتقد أن التحكّم في الفأرة أمر أكثر تعقيداً من التحكم في لوحة المفاتيح ؟
كلا، بل حتّى أن الأمر أسهل، سترى !

سترى بأن الفأرة يمكن لها أن تُنتج ثلاثة أنواع مختلفة من الأحداث.

- `SDL_MOUSEBUTTONDOWN` حينما ننقر بالفأرة، و هذا الحدث يوافق اللحظة الذي يكون فيه زر الفأرة مضغوطاً.
- `SDL_MOUSEBUTTONUP` : حينما نحرر زر الفأرة. كل هذا يعمل وفقاً لنفس المبدأ التي تعمل به أزرار لوحة المفاتيح : يوجد ضغط للزر ثم تحرير لهذا الأخير.
- `SDL_MOUSEMOTION` : حينما نقوم بتحريك الفأرة. في كل مرة تقوم فيها الفأرة بالتحرك في النافذة (هذا لا يتم إلا بيكسلا ببيكسل)، يتم إنتاج الحدث `SDL_MOUSEMOTION` !

سنبدأ أولا بالعمل على النقر على الفأرة و بشكل خاص على `SDL_MOUSEBUTTONUP`. لن نعمل مع الـ `SDL_MOUSEBUTTONDOWN`، لكنك تعرف بأن الطريقة لا تختلف إلا أن الحدث الأخير يُنتج قبل الحدث الآخر. سنعلم لاحقاً كيف نتعامل مع الحدث `SDL_MOUSEMOTION`.

معالجة نقرات الفأرة

سنقوم إذا باستقبال حدث من نوع `SDL_MOUSEBUTTONUP` (النقر بالفأرة) ثم نرى ما يمكننا استرجاعه من معلومات.
كالعادة، يجدر بنا إضافة حالة `case` في الـ `switch` كالتالي :

```
1  switch(event.type)
2  {
3          case SDL_QUIT:
4          cont = 0;
5          break;
6          case SDL_MOUSEBUTTONUP: // Mouse click
7          break;
8  }
```

لحد الآن، لا توجد صعوبة كبيرة.

ما هي المعلومات التي يمكن استرجاعها حينما ننقر بالفأرة. لدينا معلومتان :

- **الزر الذي قمنا بالضغط عليه** (الزر الأيسر ؟ الأيمن ؟ الأوسط ؟)،
- **إحداثيات مؤشر الفأرة** لحظة النقر (x و y).

استرجاع زر الفأرة

يجب أن نرى أولا أي الأزرار تم الضغط عليها . من أجل هذا، يجب تحليل المركب `event.button.button` و مقارنة قيمته بإحدى القيم التالية :

- `SDL_BUTTON_LEFT` : الضغط بالزر الأيسر للفأرة.
- `SDL_BUTTON_MIDDLE` : الضغط بالزر الأوسط للفأرة (لا يملكه كل شخص، و هو يمثل غالبا النقر بالعجلة).
- `SDL_BUTTON_RIGHT` : النقر بالزر الأيمن للفأرة.
- `SDL_BUTTON_WHEELUP` : تحريك عجلة الفأرة إلى الأعلى.
- `SDL_BUTTON_WHEELDOWN` : تحريك عجلة الفأرة إلى الأسفل.

> الثابتان الأخيران يوافقان تحريك عجلة الفأرة إلى الأعلى و الأسفل. و هما لا يوافقان "النقر" على العجلة كما يمكن أن نعتقد بالخطأ.

سنقوم باختبار سهل لنرى ما إن تم الضغط بالزر الأيمن للفأرة. إذا ضغطنا عليه، نخرج من البرنامج. (أعرف أن هذا ليس بقرار مناسب لكن لكي نجرّب لا أكثر) :

```
1  switch(event.type)
2  {
3          case SDL_QUIT:
4          cont = 0;
5          break;
6          case SDL_MOUSEBUTTONUP:
7          if (event.button.button == SDL_BUTTON_RIGHT)
8          // We stop the program on right-click with the mouse
9                  cont = 0;
10         break;
11 }
```

يمكنك التجريب، سترى بأن البرنامج يتوقف حين يتم النقر بالزر الأيمن للفأرة.

استرجاع إحداثيات الفأرة

هذه معلومة جد مهمة : إحداثيات مؤشّر الفأرة في حين النقر !
سنقوم باستعادتها بواسطة مركّبين (واحد من أجل الفاصلة و آخر من أجل الترتيبة) : `event.button.x` و `event.button.y`.

فلنستمتع قليلاً : سنقوم بلصق Zozor في الوضعية التي توافق إحداثيات النقطة التي تم النقر عليها بالفأرة.
هل هذا صعب ؟ لا أبداً ! حاول فعل ذلك، سترى بأنها لعبة أطفال !

هاهو التصحيح :

```
while (cont)
{
        SDL_WaitEvent(&event);
        switch(event.type)
        {
                case SDL_QUIT:
                cont = 0;
                break;
                case SDL_MOUSEBUTTONUP:
                zozorPosition.x = event.button.x;
                zozorPosition.y = event.button.y;
                break;
        }
        SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 255, 255, 255));
        // We put Zozor in its new position
        SDL_BlitSurface(zozor, NULL, screen, &zozorPosition);
        SDL_Flip(screen);
}
```

هذه الشفرة تشبه الشفرة التي كتبتُها من أجل أزرار لوحة المفاتيح. هنا الأمر أسهل بكثير : نضع مباشرة قيمة `x` في المتغير `zozorPosition.x` و نفس الشيء بالنسبة لـ `y`.
ثم نقوم بلصق Zozor في الإحداثيات الخاصة به، و هاهي النتيجة :



إليك تمريناً سهلاً جداً : لحد الآن، نقوم بتحريك Zozor مهما كان زر الفأرة الذي قمنا بضغطه. حاول ألا تحرّكه إلا إذا كان الزر المضغوط هو الأيسر. إذا تم الضغط على الزر الأيمن، يتوقف البرنامج.

معالجة تحرّكات الفأرة

تحرّك الفأرة يقوم بإنتاج حدث من نوع `SDL_MOUSEMOTION`.
و تيقن أنه يتم إنتاج أحداث بالقدر الذي تحركه به الفأرة بيكسلا ببيكسل في الشاشة ! لو نحرّك الفأرة 100 بيكسل (هذا ليس كثيرا)، سيكون إذا هناك 100 حدث مُنتج.

> ؟ لكن ألا تقوم هذه العملية بإنتاج الكثير من الأحداث بالنسبة للحاسوب ؟

لا بالتأكيد، تيقّن أنه يستقبل أكثر من ذلك بكثير !

حسناً، هل ما نقوم باسترجاعه مهم هنا ؟
إحداثيات الفأرة، بالطبع ! سنجدها في `event.motion.x` و `event.motion.y`.

> ! احذر : لن نعمل بنفس المركّبين اللذين استعملناهما من أجل النقر بالفأرة قبل قليل (سابقاً، كانت `event.button.x`). المركّبات المُستعملة تكون مختلفة في الـSDL بحسب الحدث.

سنقوم بتحريك Zozor إلى نفس إحداثيات الفأرة، هنا أيضا. سترى أن العملية فعالة و بسيطة في نفس الوقت !

```
while (cont)
{
        SDL_WaitEvent(&event);
```

```
        switch(event.type)
        {
                case SDL_QUIT:
                cont = 0;
                break;
                case SDL_MOUSEMOTION:
                zozorPosition.x = event.motion.x;
                zozorPosition.y = event.motion.y;
                break;
        }
        SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 255, 255, 255));
        SDL_BlitSurface(zozor, NULL, screen, &zozorPosition);
        SDL_Flip(screen);
}
```

حرّك Zozor في الشاشة، ما رأيك ؟
سيتبع طبيعياً الفأرة أينما تحرّك. هذا أمر جميل، سريع، و مرن (بفضل الـdouble buffering).

دوال أخرى من أجل التعامل مع الفأرة

سنرى دالتين سهلتي الاستعمال لهما علاقة بالفأرة. هاتان الدالتان ستكونان مفيدتين قريباً.

إخفاء مؤشّر الفأرة

يمكننا إخفاء مؤشر الفأرة بسهولة تامة، يكفي أن نستدعي الدالة `SDL_ShowCursor` و نُعطيها عَلَماً :

- `SDL_DISABLE` : إخفاء مؤشّر الفأرة.
- `SDL_ENABLE` : إظهار مؤشر الفأرة.

مثلا :

```
SDL_ShowCursor(SDL_DISABLE);
```

مؤشّر الفأرة سيبقى مختفياً طالما هو داخل النافذة.
أنصحك بأن تقوم بإخفائه قبل الحلقة الرئيسية في البرنامج لأنه لا داعي لإخفائه في كلّ دورة للحلقة، مرة واحدة كافية.

وضع الفأرة في موضع محدد

يمكننا تحريك مؤشر الفأرة يدويا إلى الإحداثيات التي نريدها في النافذة.
نستعمل من أجل هذا `SDL_WarpMouse` و التي تأخذ كعاملين الإحداثيتان `x` و `y` أين يجدر بالمؤشّر أن يتواجد.

مثلا، الشفرة المصدرية التالية تقوم بتحريك الفأرة إلى وسط النافذة :

```
SDL_WarpMouse(screen->w / 2, screen->h / 2);
```

> حينما تستدعي `SDL_WarpMouse`، يتم إنتاج حدث من نوع `SDL_MOUSEMOTION`. نعم، فالفأرة تحرّكت حتى و إن لم يقم المستعمل بذلك، فقد كان هناك تحرّك رغم ذلك !

5.23 أحداث النافذة

النافذة نفسها يمكن لها إنتاج عدد من الأحداث :

- حينما يتم تغيير مقاييسها.
- حينما يتم إنزالها إلى شريط المهام السُفلي و حينما يتم استعادتها.
- حينما تصبح مفعّلة (شاشة أولى) أو حينما تصبح غير مفعلة.
- حينما يكون مؤشر الفأرة داخل النافذة أو حينما يخرُج منها.

فلنبدأ بدراسة الحالة الأولى : الحدثُ الذي يُنتج حينما يتم تغيير مقاييس النافذة.

تغيير مقاييس النافذة

بشكل افتراضي، تكون مقاييس النافذة غير قابلة للتعديل من طرف المستخدم.
أذكّرك بأنه لتغيير ذلك، يجب إضافة العلَم `SDL_RESIZABLE` إلى الدالة `SDL_SetVideoMode` :

```
screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF |
    SDL_RESIZABLE);
```

ما إن تتم إضافة هذا العلَم، يمكنك تغيير مقاييس النافذة. حينما تقوم بذلك، يتم إنتاج حدث من نوع `SDL_VIDEORESIZE`.

يمكنك استرجاع :

- العرَض الجديد من `event.resize.w`.

• الارتفاع الجديد من `event.resize.h`.

يمكننا استعمال هذه المعلومات من أجل أن نعمل على أن يكون Zozor متمركزاً دائما في وسط النافذة :

```
case SDL_VIDEORESIZE:
zozorPosition.x = event.resize.w / 2 - zozor->w / 2;
zozorPosition.y = event.resize.h / 2 - zozor->h / 2;
break;
```

مرأى النافذة (Window visibility)

يتم إنتاج الحدث `SDL_ACTIVEEVENT` حينما يتم تغيير مرأى النافذة.
قد يحصل هذا لأسباب كثيرة :

• حينما يتم إنزال النافذة إلى شريط المهام السفلي أو استرجاعها.

• تواجد مؤشر الفأرة داخل النافذة أو خارجها.

• حينما يتم تفعيل النافذة أو إلغاء ذلك.

م

في البرمجة نتكلم عن التركيز (Focus). حينما نقول أن تطبيقاً يملك التركيز، فهذا يعني أنه يستعمل لوحة المفاتيح أو الفأرة. و أي ضغط على أزرار لوحة المفاتيح أو نقر بالفأرة يتم إرسالها إلى النافذة التي بها التركيز و ليس لأخرى. نافذة واحدة لها الحق في أن يكون لها تركيز في لحظة معيّنة (لا يمكن أن تكون نافذتان كشاشة أولى في آن واحد !)

نظراً لكثرة الأسباب التي يمكن لها أن تسبب وقوع حدث ما، يجب قطعاً معرفة قيمة المتغيرات التالية لمعرفة المزيد :

• `event.active.gain` : تشير ما إن كان الحدث ربحا (1) أو خسارة (0). مثلاً، إذا انتقلت النافذة إلى الخلفية، فهذه خسارة (0) و إذا تم ارجاعها كشاشة أولى فهذا ربح (1).

• `event.active.state` : هو مزج بين عدة أعلام للإشارة إلى نوع الحدث الذي تم إنتاجه. هذه قائمة الأعلام الممكنة :

  – `SDL_APPMOUSEFOCUS` : مؤشّر الفأرة كان بصدد الدخول أو الخروج من النافذة. يجب رؤية قيمة `event.active.gain` لنعرف ما إن كان قد دخل (ربح = 1) أو خرج (ربح = 0).

  – `SDL_APPINPUTFOCUS` : قامت النافذة باستقبال تركيز لوحة المفاتيح أو فقده. أي أنها انتقلت للخلفية أو رجعت كشاشة أولى.
  مرة أخرى، يجب رؤية قيمة `event.active.gain` لنعرف ما إن تم وضع النافذة في الخلفية (ربح = 0) أو في الشاشة الأولى (ربح = 1).

- `SDL_APPACTIVE` : تمت عملية أيقنة النافذة، أي أنه تم إنزالها إلى شريط المهام السفلي (ربح = 0) أو إرجاعها إلى مكانها الأصلي (ربح = 1).

أمازلت تتابعني ؟ يجب مقارنة قيم المركّبات `gain` و `state` لمعرفة ما حصل بالفعل.

اختبار قيمة الأعلام المدمجة

`event.active.state` هي عبارة عن دمج للأعلام. هذا يعني أنه في حدث، يمكن أن يحصل أمران (مثلاً، لو ننزل النافذة إلى شريط المهام، سنفقد تركيز لوحة المفاتيح و الفأرة أيضاً).
لهذا، يجب القيام باختبار أكثر تعقيدا من هذا :

```
1 if (event.active.state == SDL_APPACTIVE)
```

لماذا الأمر معقد ؟

لأنه عبارة عن مزج لبيتات (bits). لن أقوم بطرح درس حول العمليات المنطقية bit بـ bit الآن، لأنّ هذا سيكون كثيرا لهذا الدرس و ليس عليك معرفة الكثير عنها.
سأقدم لك شفرة قابلة للتطبيق و التي يجب عليك استعمالها إذا وُجد عَلَمٌ في متغير دون الدخول في التفاصيل.

لتجريب ما إن تم أي تغيير في التركيز الخاص بالفأرة مثلاً، يجدر بنا كتابة :

```
1 if ((event.active.state & SDL_APPMOUSEFOCUS) == SDL_APPMOUSEFOCUS)
```

لا توجد أخطاء. إحذر، الأمر دقيق : يجب استعمال إشارة `&` واحدة و اثنتين من `=`، و يجب استعمال الأقواس كما فعلت.

الشفرة تعمل بنفس الطريقة بالنسبة للأحداث الأخرى، مثلا :

```
1 if ((event.active.state & SDL_APPACTIVE) == SDL_APPACTIVE)
```

اختبار الحالة و الربح في آن واحد

عملياً، ستحتاج مؤكّدا إلى اختبار الحالة و الربح في آن واحد. هكذا يمكنك معرفة ما قد حصل بالفعل.

لنفترض أننا نبرمج لعبة تجعل الحاسوب يقوم بالكثير من الحسابات، تريد أن يتوقف البرنامج مؤقّتا تلقائيا عندما يتم إنزال النافذة إلى شريط المهام، ثم عند إعادتها إلى وضعها الأصلي، يُكمل البرنامج عمله تلقائياً. هذا سيجنبنا الحالة التي يستمر فيها البرنامج بالعمل حتى في غياب اللاعب، و سيجنبنا أيضاً جعل المعالج يقوم بالكثير من الحسابات التي لا فائدة منها.

الشفرة التالية تسمح للبرنامج بالتوقف قليلاً و ذلك بتفعيل المتغير المنطقي `pause` (إعطائه القيمة 1). تقوم الشفرة بإكمال عمل البرنامج بتعطيل المتغير المنطقي (إعطائه القيمة 0).

```
if ((event.active.state & SDL_APPACTIVE) == SDL_APPACTIVE)
{
        if (event.active.gain == 0) // If the window is minimized
                pause = 1;
        else if (event.active.gain == 1) // If the window is restored
                pause = 0;
}
```

هذه الشفرة ليست كاملة بطبيعة الحال. عليك اختبار قيمة المتغير `pause` لمعرفة ما إن كان واجباً القيام بالحسابات في برنامجك أو لا.

سأترك لك القيام باختبارات من أجل الحالات الأخرى (مثلا، التأكد ما إن كان مؤشر الفأرة داخل أو خارج النافذة) يمكنك التدرّب و ذلك بتحريك Zozor إلى اليمين إذا دخل مؤشّر الفأرة إلى النافذة و تحريكه إلى اليسار إذا خرج المؤّشر منها.

ملخّص

- الأحداث هي عبارة عن إشارات ترسلها إلينا الـSDL من أجل إطلاعنا على فعل قام به المستعمل : الضغط على زر، تحريك أو النقر بالفأرة، غلق النافذة، إلخ.
- يتم استرجاع الأحداث في متغير من نوع `SDL_Event` بواسطة الدالة `SDL_WaitEvent` (دالة معطِّلة لكن يسهلُ التحكّم بها) أو الدالة `SDL_PollEvent` (دالة غير معطِّلة لكن يصعب التحكّم بها).
- يجب تحليل المركّب `event.type` من أجل معرفة نوع الحدث الذي تم إنتاجه. نقوم بذلك غالبا داخل `switch`.
- ما إن يتم تحديد نوع الحدث، نحتاج غالبا إلى تحليل الحدث بالتفصيل. مثلا، عند الضغط على زر في لوحة المفاتيح (`SDL_KEYDOWN`)، يجب تحليل المركّب `event.key.keysym.sym` لمعرفة الزر الذي تم الضغط عليه بالضبط.
- تقنية الـdouble buffering هي تقنية تسمح بتحميل الصورة التالية في الشاشة الخلفية و إظهارها فقط حينما تكون جاهزة. هذا يسمح بتجنب تقطّع الصورة في الشاشة.

الفصل 24

عمل تطبيقي : Mario Sokoban

المكتبة SDL تقدّم، مثلما رأينا، عددا كبيراً من الدوال الجاهزة للاستعمال. يمكن ألا نستطيع التعوّد عليها في البداية لقلّة التطبيق.

هذا العمل التطبيقي الأول في هذا الجزء من الكتاب سيعطيك فرصة التطبيق و اختبار أشياء لم تسنح لك فرصة تجريبها.
أعتقد أنه بإمكانك التخمين، فهذه المرة لن يكون التطبيق عبارة عن كونسول و إنما سيتحتوي على واجهة رسومية !

ماذا سيكون موضوع هذا العمل التطبيقي ؟ لعبة Sokoban !
قد لا يعني لك هذا العنوان شيئاً، لكن هذه هي لعبة ذكاء تقليديّة. إنّها تنصّ على دفع صناديق لوضعها في أماكن محددة في متاهة.

1.24 مواصفات Sokoban

بخصوص Sokoban

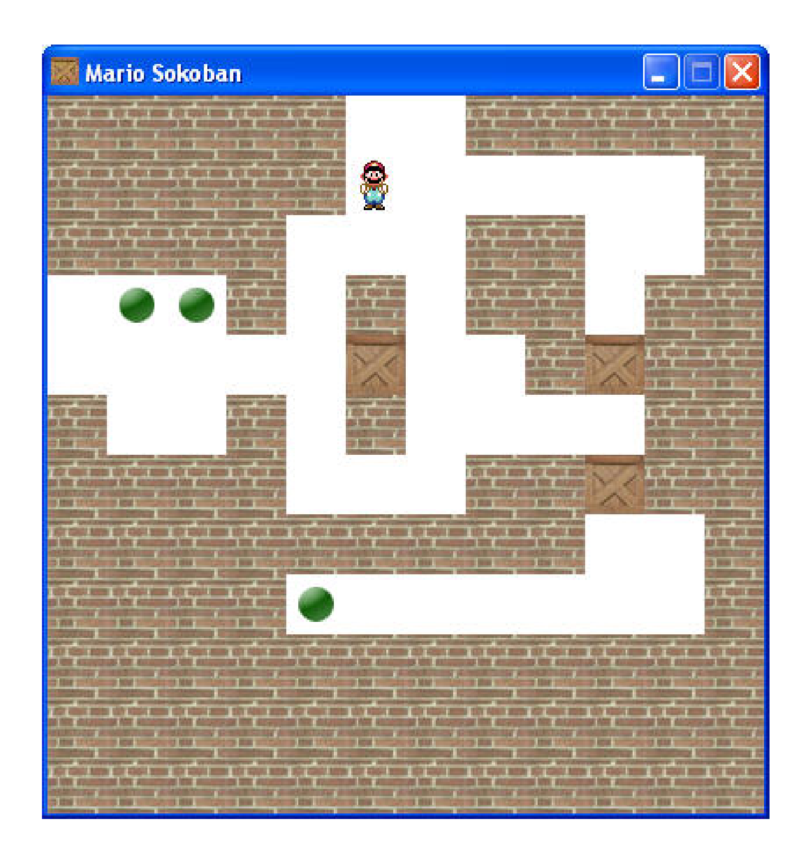
الكلمة "Sokoban" هي كلمة يابانية تعني "صاحب محلّ".
إنّها عبارة عن لعبة ذكاء تم اختراعها في الثمانينات بواسطة Hiroyuki Imabayashi. و قد مثّلت برمجة هذه اللعبة تحدّيا كبيراً في ذلك الزمن.

الهدف من اللعبة

المبدأ بسيط : تقوم بتحريك شخصية في متاهة. يجدر بالشخصية أن تقوم بدفع صناديق إلى مواقع محددة. لا يمكن للاعب أن يدفع صندوقين في آن واحد.

حتى و إن كان المبدأ مفهوماً و بسيطاً، فهذا لا يعني أن اللعبة في حدّ ذاتها سهلة ! إذ أنه يجب عليك أحيانا تكسير رأسك بالتفكير لحلّ اللغز.

الصورة الموالية تُريك كيف تبدو اللعبة التي سنقوم ببرمجتها :



لماذا اخترتُ هذه اللعبة بالذات ؟

لأنها لعبة شعبية، جيدة لأن تكون موضوعاً برمجياً و يمكننا إنشاؤها بواسطة ما تعلّمناه من الفصول السابقة.
يجب هنا أن نكون منظّمين. إذ أن الصعوبة لا تكمُن في برمجة اللعبة في حدّ ذاتها لكن في ما إن نظّمنا العمل. و لهذا فسنقوم بتقسيم البرنامج إلى عدّة ملفات `.c` بطريقة ذكيّة و نحاول إنشاء الدوال المناسبة.

من أجل هذا الأمر، قررت تغيير الطريقة بالنسبة لهذا العمل التطبيقي : لن أقدّم لك توجيهات و أقدّم التصحيح في النهاية. بالعكس، سأريك كيف نقوم ببناء المشروع كلّه من الألف إلى الياء.

؟ ماذا لو كنتُ أريد التدرّب لوحدي ؟

حسناً إذا فلتنطلق لوحدك، هذا أمر جيد !
ستحتاج ربّما وقتاً أكثر : لقد استغرقت شخصيا يوماً كاملاً لبرمجة اللعبة، هذا ليس بالوقت الكثير ربّما لأنه جرت العادة أن أقوم بالبرمجة و و أن أتحاشى الوقوع في بعض الأفخاخ المتداولة (لكنّ هذا لم يمنعني من إعادة التفكير عدّة مرّات).

إعلم بأنه توجد الكثير من الطرق التي يمكن بها برمجة هذه اللعبة. سأعطيك طريقتي في برمجتها : ليست أحسن طريقة و لكنها بالتأكيد ليست أسوء واحدة.
سننتهي من هذا التطبيق بقائمة من الإقتراحات لتحسين اللعبة، كما أنني سأعطيك الرابط لتحميل اللعبة و الشفرة المصدرية الكاملة.

أنصحك مجدداً أن تحاول برمجة اللعبة لوحدك، حتى لو استغرقت 3 أو 4 أيام. إفعل أحسن ما لديك. من المهم جدّا أن تقوم بالتطبيق.

المواصفات

المواصفات هي عبارة عن وثيقة نكتب فيها كل ما يجب على البرنامج أن يستطيع فعله.

هذا ما أقترحه :

- يجب أن يتمكن اللاعب من التحرّك في المتاهة و دفع الصناديق.
- لا يمكنه أن يدفع صندوقين معاً.
- تُربح الجولة إذا تواجدت كلّ الصناديق في الأماكن المخصصة لها.
- سيتم حفظ كلّ مستويات اللعبة في ملف، (ليكن مثلا `levels.lvl`).
- يجب أن يتم دمج مُنشئ المستويات (Levels editor) في البرنامج ليتمكن أي شخص كان من صنع مستويات خاصة به (هذا ليس أمراً ضرورياً لكنه يعتبر إضافة مميزة !).

هذا كافٍ لنعمل كثيراً.

يجب أن تعرف أنه هناك أشياء لا يجيد البرنامج القيام بها، و يجب ذِكرُ هذا الأمر أيضاً.

- برنامجنا قادر على التحكّم في مرحلة واحدة في المرّة الواحدة. إن أردت أن تكون اللعبة عبارة عن تتالي جولات، فما عليك سوى برمجة ذلك بنفسك في نهاية هذا العمل التطبيقي.
- البرنامج لا يقوم بحساب الوقت المُستغرق في كلّ جولة (نحن لا نجيد فعل ذلك بعد) و لا يمكنه حساب النقاط.

على أي حال، فكلّ الأشياء التي نريد القيام بها (خاصة مُنشِئ المراحل) تأخذ منا وقتاً لا بأس به.

> م
>
> سأعطيك في نهاية العمل التطبيقي، جملة التحسينات التي تُمكن إضافتها إلى اللعبة. و هذه ليست كلمات في الهواء، لأنّها أفكار طبّقتها أنا شخصيّا في نسخة كاملة من اللعبة سأقترح عليك تنزيلها.
> بالمقابل، لن أعطيك الشفرة المصدرية الخاصة بالنسخة الكاملة لأنني أريدك أن تعمل بنفسك و تتدرّب (لن أعطيك كلّ شيء على طبق من فضّة !).

الحصول على الصور اللازمة للعبة

في معظم الألعاب ثنائية الأبعاد، أيّا كان نوعها، نسمّي الصور التي تشكّل اللعبة *Sprites*.
في حالتنا، قرّرت إنشاء Sokoban و وضع الشخصية Mario لتكون اللاعب الرئيسي فيها (من هنا جاء اسم اللعبة Mario

Sokoban). بما أن Mario شخصية لها شعبية كبيرة في عالم الألعاب 2D، لن نتعب في الحصول على الـsprites الخاصّة بهذه الشخصيّة. سنحتاج أيضا إلى sprites خاصة بالجدران، الصناديق، الأماكن المستهدفة، إلخ.

إذا بحثت في Google عن "sprites" فستحصل على عدّة نتائج. توجد العديد من المواقع التي توفّر sprites خاصّة بألعاب 2D قد تكون لعبتها في السابق.

و هذه هي الّتي سنحتاج إليها :

Sprite	الشرح
	جدار
	صندوق
	صندوق متموضع فوق منطقة مستهدفة
	بطل اللعبة (Mario) باتجاه الأسفل
	بطل اللعبة باتجاه اليمين
	بطل اللعبة باتجاه اليسار
	بطل اللعبة باتجاه الأعلى

الأسهل هو أن تقوم بتحميل الحزمة التي أعددتها لك.

https://openclassrooms.com/uploads/fr/ftp/mateo21/sprites_mario_sokoban.zip

كان من الممكن أن أستعمل sprite واحداً خاصاً باللاعب. كان بإمكاني جعله موجّهاً إلى الأسفل فقط، لكن إضافة امكانية توجيهه في الاتجاهات الأربعة تضيف القليل من الواقعية. و هذا يشكّل تحدّيا آخر لنا !

قمت أيضاً بإنشاء صورة أخرى لتكون عبارة عن الواجهة الأساسية للعبة حين تبدأ، لقد أرفقت لك الصورة بالحزمة الّتي يفترض بك تنزيلها. لاحظ الصورة التالية :



ستلاحظ بأن الصور تأخذ صيغا مختلفة. يوجد منها ماهو GIF، ماهو PNG و حتى ماهو JPEG. و لهذا فنحن بحاجة إلى استعمال المكتبة SDL_Image.
فكّر في جعل مشروعك يعمل مع الـSDL و الـSDL_Image. إذا نسيت كيف تفعل ذلك، فراجع الفصول السابقة. إذا لم تقم بتخصيص المشروع بشكل صحيح، سيشير المُترجم بأن الدوال التي تستعملها (مثل `IMG_Load`) غير موجودة !

2.24 الدالة main و الثوابت

في كلّ مرة نبدأ بتحقيق مشروع مهمّ، من الواجب أن نقوم بتنظيم العمل في البداية.
بشكل عام، أبدأ في إنشاء ملف ثوابت `constants.h` إضافة إلى ملف `main.c` يحتوي الدالة `main` (فقط هذه الدالة). هذه ليست قاعدة لكنها طريقتي الخاصة في العمل، و لكلّ شخص طريقته الخاصة.

ملفّات المشروع المختلفة

أقترح أن نقوم بإنشاء ملفات المشروع كلّه الآن، (حتى و إن كانت فارغة في البداية). هاهي الملفات التي أنشئها إذا :

- `constants.h` : تعريف الثوابت الشاملة الخاصة بكل البرنامج.
- `main.c` : الملف الذي يحتوي `main` (الدالة الرئيسية في البرنامج).
- `game.c` : الدوال التّي تسيّر جولة من اللعبة Sokoban.
- `game.h` : نماذج الدوال الخاصة بالملف `game.c`.
- `editor.c` : ملف يحتوي الدول التي تحكم في مُنشئ المستويات.
- `editor.h` : نماذج الدوال الخاصة بالملف `editor.c`.
- `files.c` : الدوال الخاصّة بقراءة و كتابة ملفّات المستويات (مثل `levels.lvl`).

• نماذج الدوال الخاصة بالملف `files.c`.

سنبدأ بإنشاء ملف الثوابت.

الثوابت : constants.h

هذا محتوى الملف `constants.h` الخاص بي :

```
/*
constants.h
------------

By mateo21, for "Site du Zéro" (www.siteduzero.com)

Role : define some constants for all of the program (window size...)
*/
#ifndef DEF_CONSTANTS
#define DEF_CONSTANTS
#define BLOCK_SIZE 34 // Block size (square) in pixels
#define NB_BLOCKS_WIDTH 12
#define NB_BLOCKS_HEIGHT 12
#define WINDOW_WIDTH BLOCK_SIZE * NB_BLOCKS_WIDTH
#define WINDOW_HIGHT BLOCK_SIZE * NB_BLOCKS_HEIGHT
enum {UP, DOWN, LEFT, RIGHT};
enum {EMPTY, WALL, BOX, GOAL, MARIO, BOX_OK};
#endif
```

ستلاحظ الكثير من النقاط المهمّة في هذا الملف الصغير.

• يبدأ الملف بتعليق رأسي. أنصحك بوضع تعليق مماثل في كلّ ملفاتك (مهما كانت صيغتها `.c` أو `.h`). بشكل عام، التعليق الرأسي يحوي :
  – اسم الملف،
  – اسم الكاتب (المبرمج)،
  – مهمّة الملف (أي فائدة الدوال الّتي يحويها)،
  – لم أقم بهذا هنا، لكن عادة يفترض أيضاً إضافة تاريخ كتابة الملف و تاريخ آخر تعديل عليه. هذا يسمح لك بإيجاد المعلومات بسرعة حينما تحتاج إليها و خاصة حينما يتعلّق الأمر بمشاريع كبيرة.

• الملف محمّي ضد التضمينات غير المنتهية. لقد استعملت لذلك التقنية التي تعلّمناها في نهاية فصل المعالج القبلي. هنا، الحماية ليست مهمّة جدّا، لكن جرت العادة أن أستعملها في كلّ ملفاتي `.h` بدون استثناء.

• أخيراً، قلب الملف. ستجد لائحة من `#define`. قمت بتحديد حجم كتلة بالبيكسل (كل sprites هي عبارة عن مربّعات ذات حجم 34 بيكسل). أحدد بأن حجم النافذة يساوي 12*12 كتلة كعُرض. و بهذا أقوم بحساب أبعاد

النافذة بعملية ضرب ثوابت بسيطة. ما أقوم به هنا ليس ضرورياً، لكنه يعود علينا بالفائدة : إذا أردت لاحقاً مراجعة حجم اللعبة، يكفي أن أقوم بتعديل هذا الملف و إعادة ترجمة المشروع فيعمل مع القيم الجديدة دون أية مشاكل.

- أخيراً، قمت بتعريف ثوابت عن طريق تعدادات غير معرّفة، الأمر مختلف قليلاً عمّا تعلّمناه في فصل إنشاء أنواع خاصة بنا. هنا أنا لست أقوم بتعريف نوع خاص بي بل أقوم فقط بتعريف ثوابت. هذا يشبه المعرّفات مع اختلاف بسيط : الحاسوب هو من يقوم بإعطاء عدد لكلّ قيمة (بدءً من 0). و بهذا يكون لدينا : `UP` = 0، `DOWN` = 1، `LEFT` = 2، إلخ. هذا ما سيسمح للشفرة بأن تكون مفهومة لاحقاً، سترى ذلك !

باختصار، لقد استعملت :

- معرّفات حينما أريد أن أعطي قيمة محددة لثابت (مثلاً 34 بيكسل).
- تعدادات حينما تكون قيمة الثابت لا تهمّني. هنا، لا يهمني ما إن كانت القيمة المُرفقة بالعنصر `UP` هي 0 (كان من الممكن أن تكون 150، هذا لن يغيّر شيئا)، كلّ ما يهمّني هو أن يكون هذا العنصر مختلفا عن `DOWN` و `LEFT` و `RIGHT`.

تضمين تعريفات الثوابت

المبدأ ينص على تضمين ملف الثوابت في كلّ الملفات `.c`.
هكذا، أستطيع استعمال الثوابت في أي مكان من الشفرة المصدرية الخاصة بالمشروع.

يعني أنه عليّ أن أكتب السطر التالي في كل بداية للملفات `.c` :

```
1 #include "constants.h"
```

الدالة main : main.c

الدالة الرئيسيّة الخاصة بالبرنامج سهلة جداً. هي تقوم بإظهار واجهة اللعبة ثم التوجيه إلى القِسم المناسب.

```
1  /*
2  main.c
3  ------
4
5  By mateo21, for "Site du Zéro" (www.siteduzero.com)
6
7  Role : game menu. Allow to choose between the editor and the game.
8  */
9  #include <stdlib.h>
10 #include <stdio.h>
11 #include <SDL/SDL.h>
12 #include <SDL/SDL_image.h>
13 #include "constants.h"
```

```
14 #include "game.h"
15 #include "editor.h"
16 int main(int argc, char *argv[])
17 {
18         SDL_Surface *screen = NULL, *menu = NULL;
19         SDL_Rect menuPosition;
20         SDL_Event event;
21         int cont = 1;
22         SDL_Init(SDL_INIT_EMPTYO);
23         SDL_WM_SetIcon(IMG_Load("box.jpg"), NULL); // The icon must be loaded
               before SDL_SetVideoMode
24         screen = SDL_SetVideoMode(WINDOW_WIDTH, WINDOW_HIGHT, 32,SDL_HWSURFACE
               | SDL_DOUBLEBUF);
25         SDL_WM_SetCaption("Mario Sokoban", NULL);
26         menu = IMG_Load("menu.jpg");
27         menuPosition.x = 0;
28         menuPosition.y = 0;
29         while (cont)
30         {
31                 SDL_WaitEvent(&event);
32                 switch(event.type)
33                 {
34                         case SDL_QUIT:
35                         cont = 0;
36                         break;
37                         case SDL_KEYDOWN:
38                         switch(event.key.keysym.sym)
39                         {
40                                 case SDLK_ESCAPE: // Want to quit the game
41                                 cont = 0;
42                                 break;
43                                 case SDLK_KP1: // Want to play
44                                 play(screen);
45                                 break;
46                                 case SDLK_KP2: // Want to edit levels
47                                 editor(screen);
48                                 break;
49                         }
50                         break;
51                 }
52                 // Cleaning the screen
53                 SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 0, 0,0));
54                 SDL_BlitSurface(menu, NULL, screen, &menuPosition);
55                 SDL_Flip(screen);
56         }
57         SDL_FreeSurface(menu);
58         SDL_Quit();
59         return EXIT_SUCCESS;
60 }
```

الدالة `main` تتكفّل بتهيئة الـSDL، و إعطاء عنوان للنافذة إضافة إلى منحها أيقونة. في نهاية الدالة، يتم استدعاء الدالة `SDL_Quit` لإيقاف الـSDL بشكل سليم.

الدالة تقوم بإظهار قائمة يتم تحميلها بواسطة الدالة `IMG_Load` من المكتبة SDL_Image.

```
menu = IMG_Load("menu.jpg");
```

تلاحظ أنه، لكي أعطي أبعاداً للنافذة، أستعمل الثابتين `WINDOW_WIDTH` و `WINDOW_HIGHT` المعرّفين في الملف `constants.h`.

حلقة الأحداث

الحلقة غير المنتهية تعالج الأحداث التالية :

- **إيقاف البرنامج** (`SDL_QUIT`) : إذا قمنا بطلب غلق البرنامج (النقر على العلامة `X` أعلى يمين النافذة) فسنعطي القيمة 0 للمتغير `cont` و نتوقف الحلقة. باختصار، هذا أمر تقليديّ.
- **الضغط على الزر** `Escape` : إغلاق البرنامج (مثل `SDL_QUIT`).
- **الضغط على الزر** `1` **من لوحة الأرقام** : انطلاق تشغيل اللعبة (استدعاء الدالة `play`).
- **الضغط على الزر** `2` **من لوحة الأرقام** : انطلاق تشغيل مُنشئ المراحل (استدعاء الدالة `editor`).

كما ترى فالأمور تجري بسهولة تامة. إذا ضغطنا على الزر 1، يتم تشغيل اللعبة، ما إن تنتهي اللعبة، تنتهي الدالة `play` و نرجع للـ`main` من أجل القيام بدورة أخرى للحلقة. الحلقة تستمر في الاشتغال مادمنا لم نطلب إيقاف البرنامج.

بفضل هذا التنظيم البسيط جدّاً، يمكننا التحكم في الدالة `main` و ترك الدوال الأخرى (مثل `play` و `editor`) تهتم بالتحكم في مختلف أجزاء اللعبة.

3.24 اللعبة

فلندخل إلى المرحلة الأكثر أهمية في الموضوع : الدالة `play` !
هذه هي الدالة الأكثر أهمية في البرنامج، كن متيقّظاً لأن هذه الدالة هي حقّا ما يجدر بك فهمه. لأنك ستجد بعدها بأن مُنشئ المراحل ليس بالصعوبة التي تخيّلها.

المعاملات التي نبعثها للدالة

الدالة `play` تحتاج إلى معامل واحد : المساحة `screen`. بالفعل، تم فتح النافذة في الدالة الرئيسية، و لكي تستطيع الدالة `play` أن ترسم على النافذة، يجب أن تقوم باسترجاع المؤشّر نحو المساحة `screen` !

لو تقرأ مجدداً محتوى الدالة الرئيسية، ستجد بأنني قمت باستدعاء الدالة `play` و ذلك بإعطائها المؤشّر `screen` :

```c
play(screen);
```

نموذج الدالة، الذي يمكنك وضعه في الملف `game.h` ، هو التالي :

```c
void play(SDL_Surface* screen);
```

الدالة لا تقوم بإرجاع أي شيء (و من هنا الـ `void`). يمكننا أن نجعلها إن أردنا تُرجع قيمة منطقية تشير إلى ما كنّا قد ربحنا الجولة أم لا.

التصريح عن المتغيرات

تحتاج هذه الدالة إلى كثير من المتغيرات.
لم أفكّر في كلّ المتغيرات التي أحتاجها من الوهلة الأولى. هناك من أضفتها لاحقاً و أنا أكتب الشفرة.

متغيّرات من أنواع معرّفة في الـSDL

لكي نبدأ، هاهي كلّ المتغيرات ذات الأنواع المعرّفة في الـSDL التي نحن بحاجة إليها :

```c
SDL_Surface *mario[4] = {NULL}; // 4 surfaces for the 4 directions of mario
SDL_Surface *wall = NULL, *box = NULL, *boxOK = NULL, *objective = NULL, *level
    = NULL, *currentMario = NULL;
SDL_Rect position, playerPosition;
SDL_Event event;
```

لقد قمتُ بإنشاء جدول من نوع `SDL_Surface` يسمّى `mario` . و هو جدول من أربع خانات يقوم بتخزين Mario في كلّ من الاتجاهات الأربعة (واحد للأسفل، الأعلى، اليمين و اليسار).

توجد بعد ذلك العديد من المساحات الموافقة لكلّ من sprites التي قمت بتحميلها أعلاه : `wall` ، `box` ، `boxOK` و `objective` .

بماذا ينفعنا `currentMario` ؟

هو عبارة عن مؤشّر نحو مساحة. و هو مؤشّر يؤشّر نحو المساحة الموافقة لـMario المتّجه نحو الإتجاه الحالي. أي أنه عبارة عن `currentMario` (Mario الحالي) الذّي سنقوم بتسويته في الشاشة. إذا رأيت في أسفل الدالة `play` ستجد :

```c
SDL_BlitSurface(currentMario, NULL, screen, &position);
```

لا نقوم إذا بلصق عنصر من الجدول `mario`، بل المؤشّر `currentMario`.
و بلصق `currentMario`، يعني أننا سنلصق إما Mario نحو الأسفل، أو نحو الأعلى، إلخ.
المؤشّر `currentMario` يؤشّر نحو إحدى خانات الجدول `mario`.

ماذا بعد غير هذا ؟
متغير `position` من نوع `SDL_Rect` سنستعين به من أجل تعريف موضع العناصر التي سنقوم بتسويتها (سنحتاج إليها من أجل كلّ الـsprites، و لا داعي لإنشاء `SDL_Rect` من أجل كلّ مساحة !). المتغير `playerPosition` مختلف : إنه يشير إلى أية خانة من الخريطة يوجد اللاعب. أخيراً، المتغير `event` يهتم بتحليل الأحداث.

متغيّرات "تقليديّة"

حان الوقت لكي أعرّف متغيرات تقليديّة نوعاً ما من نوع `int` :

```
int cont = 1, remainingGoals = 0, i = 0, j = 0;
int map[NB_BLOCKS_WIDTH][NB_BLOCKS_HEIGHT] = {0};
```

`cont` و `remainingGoals` هي متغيرات منطقية.
`i` و `j` هي متغيرات مُساعِدة ستساعدنا في قراءة الجدول `map`.

هنا تبدأ الأمور الهامّة حقّاً. لقد قمت فعلياً بإنشاء جدول ذو بُعدين. لم أكلّمك عن هذا النوع من الجداول من قبل، لكنه الوقت المناسب لتتعلّم ما يعنيه. ليس الأمر صعباً، سترى ذلك بنفسك.

لاحظ التعريف عن كثب :

```
int map[NB_BLOCKS_WIDTH][NB_BLOCKS_HEIGHT] = {0};
```

هو عبارة عن جدول من `int` (أعداد صحيحة) يختلف في كونه يأخذ حاضنتين مربّعتين `[ ]`. إذا كنت تتذكر جيداً الملف `constants.h`، فـ `NB_BLOCKS_WIDTH` و `NB_BLOCKS_HEIGHT` هما ثابتان يأخذ كلاهما القيمة 12.

هذا الجدول سيتمّ إنشاءه في وقت الترجمة هكذا :

```
int map[12][12] = {0};
```

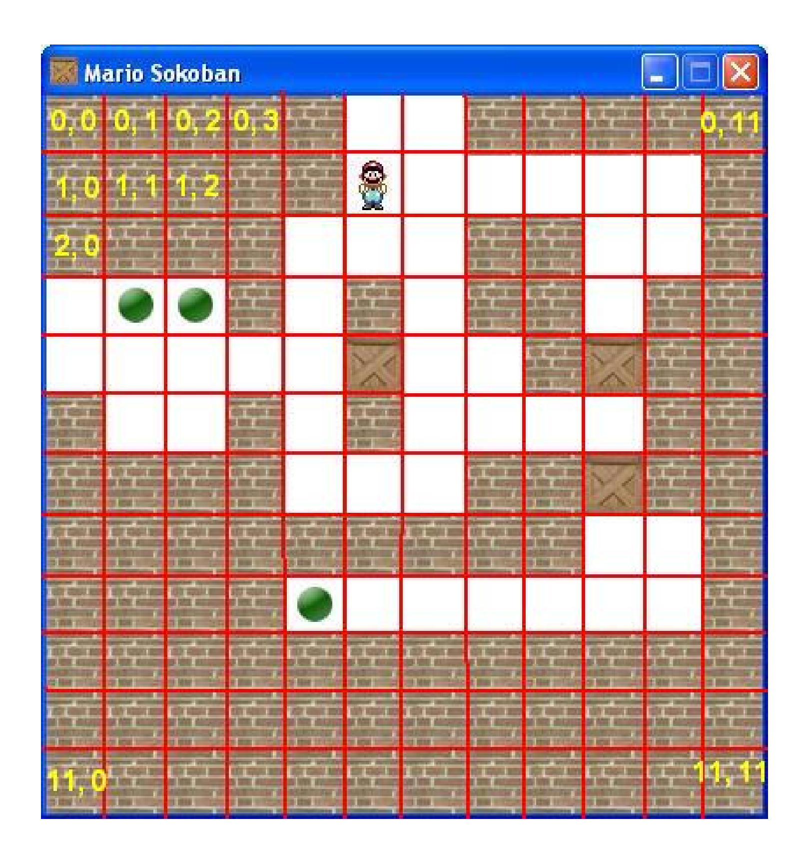
> ؟ لكن، ماذا يعني هذا ؟

هذا يعني أنه من أجل كلّ "خانة" من `map` توجد 12 خانة داخلية.
بهذا تكون لدينا المتغيرات التالية :

```
map[0][0]
map[0][1]
map[0][2]
map[0][3]
map[0][4]
map[0][5]
map[0][6]
map[0][7]
map[0][8]
map[0][9]
map[0][10]
map[0][11]
map[1][0]
map[1][1]
map[1][2]
map[1][3]
map[1][4]
map[1][5]
map[1][6]
map[1][7]
map[1][8]
map[1][9]
map[1][10]
...
map[11][2]
map[11][3]
map[11][4]
map[11][5]
map[11][6]
map[11][7]
map[11][8]
map[11][9]
map[11][10]
map[11][11]
```

إذا هو جدول من 12 * 12 = 144 خانة !
كلّ من هذه الخانات تمثّل خانة في خريطة اللعبة.

الصورة التالية تعطيك فكرة كيف قُمنا بتمثيل الخريطة :



و بهذا فالخانة في أعلى اليسار مخزّنة في `map[0][0]`.
الخانة أعلى اليمين مخزنة في `map[0][11]`.
الخانة أسفل اليمين (آخر خانة) مخزنة في `map[11][11]`.

حسب قيمة الخانة (و التي هي عدد صحيح)، نعرف أي خانة من النافذة تحتوي جداراً، أو صندوقاً، أو منطقة مستهدفة، إلخ.
هنا بالضبط سنستفيد من تعريف التعداد السابق !

```
enum {EMPTY, WALL, BOX, GOAL, MARIO, BOX_OK};
```

إذا كانت قيمة الخانة تساوي `EMPTY` (0) سنعرف بأن هذه المنطقة من الشاشة يجب أن تبقى بيضاء. إذا كانت تساوي `WALL` (1) فسنعرف أنه يجب أن نقوم بلصق صورة جدار، إلخ.

تهيئات

تحميل المساحات

و الآن، بما أننا قمنا بشرح كل متغيرات الدالة `play`، يمكننا البدء في القيام ببعض التهيئات :

```
// Loading the sprites (Boxes, player...)
wall = IMG_Load("wall.jpg");
box = IMG_Load("box.jpg");
boxOK = IMG_Load("box_ok.jpg");
level = IMG_Load("level.png");
```

```
6 mario[DOWN] = IMG_Load("mario_bas.gif");
7 mario[LEFT] = IMG_Load("mario_gauche.gif");
8 mario[UP] = IMG_Load("mario_haut.gif");
9 mario[RIGHT] = IMG_Load("mario_droite.gif");
```

لا يوجد شيء صعب : نقوم بتحميل الكلّ بواسطة `IMG_Load`.
إن كانت هناك حالة خاصّة، فهي تحميل Mario. إذ أننا نقوم بتحميل Mario في كلّ من الإتّجاهات الأربعة في الجدول `mario` باستعمال الثوابت : `UP`، `DOWN`، `LEFT`، `RIGHT`. كوننا استعملنا هنا ثوابت فستصبح الشفرة أكثر وضوحاً - كما تلاحظ- . كان بإمكاننا استعمال `mario[0]`، لكن من الأفضل و من الأكثر وضوحاً أن نستعمل `mario[UP]` مثلاً !

التوجيه الابتدائي لـMario (currentMario)

نهيّئ بعدها `currentMario` لكيّ تكون له وجهة ابتدائيّة :

```
1 currentMario = mario[DOWN]; // Mario will be headed down when starting the
      program
```

وجدت أنه من المنطقي أكثر أن أبدأ المرحلة فيما يكون Mario موجّهاً نحو الأسفل (أي نحونا)، كان بامكانك أن تكتب مثلاً :

```
1 currentMario = mario[RIGHT];
```

ستُلاحظ بأن Mario سيكون موجّهاً نحو اليمين في بداية اللعبة.

تحميل الخريطة

الآن، يجدر بنا ملء الجدول ثنائي الأبعاد `map`. لحدّ الآن، الجدول لا يحتوي إلا أصفاراً.
يجب أن نقرأ المستوى المخزّن في الملف `levels.lvl` :

```
1 // Loading the level
2 if (!loadLevel(map))
3         exit(EXIT_FAILURE); // We stop the game if we couldn't load the level
```

لقد اخترت معالجة تحميل (و حفظ) المستويات بواسطة دوال متواجدة بالملف `files.c`.
هنا ، نستدعي إذا الدالة `loadLevel`. سنقوم بدراستها بالتفصيل لاحقا (هي ليست معقدة كثيراً على أي حال). كل ما يهمنا هنا هو معرفة أنه تم تحميل المستوى في الجدول `map`.

> X
> إذا لم يتم تحميل المستوى (لأن ملف `levels.lvl` غير موجود)، ستُرجع الدالة "خطأ". أمّا في الحالة المعاكسة فتُرجع "صحيح".

نقوم إذا باختبار نتيجة التحميل بواسطة شرط. إذا كانت النتيجة سلبية (من هنا استعملت إشارة التعجّب لأعبّر عن ضدّ الشرط) يتوقف كلّ شيء : سنستدعي الدالة `exit` .
في الحالة الأخرى، كلّ شيء يعمل بشكل جيّد إذاً و يمكننا المواصلة.

نحن نملك الآن جدولا `map` يصف محتوى كلّ خانة : `WALL`، `EMPTY`، `BOX`...

البحث عن وضعية الانطلاق لـMario

يجب الآن أن نعطي قيمة ابتدائية للمتغير `playerPosition` .
هذا المتغير من نوع `SDL_Rect` خاصّ قليلاً. لن نستعين به لتخزين الإحداثيات بالبيكسل و إنما بتخزينه بدلالة الـ"خانات" في الخريطة. و بهذا فإن كانت لدينا :

`playerPosition.x == 11` و `playerPosition.y == 11`

فهذا يعني أن اللاعب متواجد في آخر خانة في أسفل يمين الخريطة.
يمكنك الرجوع إلى الصورة السابقة لتتوضح لك الأمور أكثر.

سنقوم بالتقدّم داخل الجدول `map` و ذلك باستعمال حلقتين. نستعمل المتغير `i` للتقدّم في الجدول عمودياً، و نستعمل المتغير `j` للتقدّم فيه أفقياً :

```
1  // We search for the position of Mario in the beginning of the game
2  for (i = 0 ; i < NB_BLOCKS_WIDTH ; i++)
3  {
4          for (j = 0 ; j < NB_BLOCKS_HEIGHT ; j++)
5          {
6                  if (map[i][j] == MARIO) // If Mario is in this position
7                  {
8                          playerPosition.x = i;
9                          playerPosition.y = j;
10                         map[i][j] = EMPTY;
11                 }
12         }
13 }
```

في كلّ خانة، نختبر ما إن كانت هذه الأخيرة تحتوي `MARIO` (أي نقطة انطلاق اللاعب في الخريطة). إذا كانت كذلك، نقوم بتخزين الإحداثيات الحالية (المتواجدة في `i` و `j`) في المتغير `playerPosition` .
نمسح أيضاً الخانة و ذلك بإعطائها القيمة `EMPTY` لكي يتم اعتبارها كخانة فارغة لاحقاً.

تفعيل تكرار الضغط على الأزرار

آخر شيء، أمر سهل جداً : سنقوم بتفعيل تكرار الضغط على الأزرار لكي نستطيع التحرّك في الخريطة بترك الزر مضغوطاً.

```
1  // Enabeling keys repetition
2  SDL_EnableKeyRepeat(100, 100);
```

الحلقة الرئيسية

حسناً، لقد قُمنا بتهيئة كلّ شيء، يمكننا الآن العمل على الحلقة الرئيسية.

إنها حلقة تقليديّة تعمل بنفس المخطط الذي تعمل به الحلقات التي رأيناها لحدّ الآن. هي فقط كبيرة قليلاً.

فلنرى عن قرب الـ `switch` الذي يختبر الحدَث :

```
switch(event.type)
{
        case SDL_QUIT
        cont = 0;
        break;
        case SDL_KEYDOWN:
        switch(event.key.keysym.sym)
        {
                case SDLK_ESCAPE:
                cont = 0;
                break;
                case SDLK_UP:
                currentMario = mario[UP];
                movePlayer(map, &playerPosition, UP);
                break;
                case SDLK_DOWN:
                currentMario = mario[DOWN];
                movePlayer(map, &playerPosition, DOWN);
                break;
                case SDLK_RIGHT:
                currentMario = mario[RIGHT];
                movePlayer(map, &playerPosition, RIGHT);
                break;
                case SDLK_LEFT:
                currentMario = mario[LEFT];
                movePlayer(map, &playerPosition, LEFT);
                break;
        }
        break;
}
```

إذا ضغطنا على الزر `Esc` ، فستنتهي اللعبة و نرجع للقائمة الرئيسية.

كما ترى، لا توجد العديد من الأحداث لنعالجها : سنختبر فقط ما إن ضغط اللاعب على الأزرار "أعلى"، "أسفل"، "يمين" أو "يسار" من لوحة المفاتيح.
على حسب الزر المضغوط نغيّر اتجاه Mario. و هنا يتدخّل المتغير `currentMario` ! إذا ضغطنا السهم الموجه نحو الأعلى إذاً :

```
currentMario = mario[UP];
```

إذا ضغطنا على السهم الموجّه نحو الأسفل فإذاً :

```
currentMario = mario[DOWN];
```

الآن، شيء مهمّ جداً : نستدعي الدالة `movePlayer`. هذه الدالة ستقوم بتحريك اللاعب في الخريطة إن كان له الحق في فعل ذلك.

- مثلاً، لا يمكننا أن نُحرك Mario إلى الأعلى إن كان متواجداً أصلاً في الحافة العلوية للنافذة.
- لا يمكننا أيضاً أن نحرّكه للأعلى إن كان فوقه جدار.
- لا يمكننا أن نحرّكه للأعلى إن كان فوقه صندوقان.
- على العكس، يمكننا تحريكه للأعلى إن تواجد صندوق واحد فوقه.
- لكن احذر، لا يمكننا تحريكه للأعلى إن تواجد صندوق واحد فوقه و كان هذا الصندوق متواجد أصلاً في الحافة العلوية للنافذة !

؟

يا إلاهي ! ماهذا السوق ؟

هذا ما نسمّيه بـ**معالجة الاصطدامات** (Collisions management). و لكي أضمن لك، نحن نقوم بالتعامل مع الاصطدامات البسيطة بما أن اللاعب يتحرّك خانة بخانة و في أربع اتجاهات فقط. في لعبة ثنائية الأبعاد أين يتحرّك اللاعب في كلّ الاتجاهات بيكسلا ببيكسل، يكون التحكّم في الاصطدامات أمرا أصعب.

لكن هناك ماهو أسوء : الألعاب ثلاثية الأبعاد. التحكم في الإصطدامات في لعبة ثلاثية الأبعاد يُعدّ كابوساً بالنسبة للمبرمجين. لحسن الحظ، توجد مكتبات للتحكّم في الاصطدامات في العوالم ثلاثية الأبعاد و التي تقوم بالكثير من العمل في مكاننا.

لنرجع للدالة `movePlayer` و لنركّز. نقوم بإعطائها ثلاثة معاملات :

- الخريطة : لكي تستطيع قراءتها و أيضاً التعديل عليها إذا قمنا بتحريك صندوق مثلاً.
- وضعية اللاعب : هنا أيضاً، يجب على الدالة قراءة و "ربما" تعديل وضعية اللاعب.
- الإتجاه الذي نطلب من اللاعب التوجّه إليه : نستعمل هنا أيضاً الثوابت : `UP`، `DOWN`، `LEFT`، `RIGHT` من أجل فهم الشفرة بشكل أفضل.

سندرس الدالة `movePlayer` لاحقاً. كان بإمكاني وضع كلّ الاختبارات داخل الـ`switch`، لكن بهذا سيصبح كبيراً و ستصعب علينا قراءته. و من هنا نرى الفائدة من تقسيم الشفرة إلى عدّة دوال.

التسوية، فلنسوّي كلّ شيء

لقد انتهينا من الـ `switch` : في هذه الوضعية من البرنامج، قد تكون الخريطة قد تغيّرت و كذا وضعية اللاعب. مهما كان، لقد حان وقت التسوية !

سنبدأ بمسح الشاشة و ذلك بإعطائها لون خلفية أبيض :

```c
// Clearing the screen
SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 255, 255, 255));
```

و الآن، نقوم بالتقدّم في الجدول ذو البعدين `map` لكي نعرف أي عنصر سنقوم بتسويته و في أي منطقة من الشاشة. سنستعمل حلقتين كما رأينا سابقاً للتقدّم في الـ144 خانة من الجدول :

```c
// Placing the objects on the screen
remainingGoals = 0;
for (i = 0 ; i < NB_BLOCKS_WIDTH ; i++)
{
        for (j = 0 ; j < NB_BLOCKS_HEIGHT ; j++)
        {
                position.x = i * BLOCK_SIZE;
                position.y = j * BLOCK_SIZE;
                switch(map[i][j])
                {
                        case WALL:
                        SDL_BlitSurface(wall, NULL, screen, &position);
                        break;
                        case BOX:
                        SDL_BlitSurface(box, NULL, screen, &position);
                        break;
                        case BOX_OK:
                        SDL_BlitSurface(boxOK, NULL, screen, &position);
                        break;
                        case GOAL:
                        SDL_BlitSurface(level, NULL, screen, &position);
                        remainingGoals = 1;
                        break;
                }
        }
}
```

من أجل كل خانة، نحضّر المتغير `position` (من نوع `SDL_Rect`) لكي نضع العنصر الحالي في المكان المناسب من الشاشة.
العملية جدّ بسيطة :

```c
position.x = i * BLOCK_SIZE;
position.y = j * BLOCK_SIZE;
```

يكفي ضرب `i` بـ `BLOCK_SIZE` لكي نعرف قيمة `position.x`.
و بهذا، فإن كنا نتواجد في الخانة الثالثة، أي أن `i` = 2 (لا تنس أن `i` يبدأ من 0 !)، نقوم إذاً بالعملية 2 * 34 = 68.
إذا نقوم بلصق الصورة 68 بيكسلا نحو اليمين في المساحة `screen`.
نقوم بنفس الشيء بالنسبة للترتيبة `y`.

بعد ذلك، نطبّق `switch` على الخانة التي نقوم بتحليلها من الخريطة.
هنا أيضاً، استعمال الثوابت يعتبر شيئاً عملياً و يسمح بقراءة مُثلى للشفرة.
نختبر إذا ما إن كانت الخانة تساوي `WALL`، في هذه الحالة نقوم بلصق جدار. نفس الشيء بالنسبة للصناديق و المناطق المُستهدفة.

اختبار الفوز

تلاحظ أنه قبل استعمال الحلقتين المتداخلتين، نعطي القيمة الإبتدائية 0 للمتغير المنطقي `remainingGoals`.
هذا المتغير المنطقي يأخذ القيمة 1 ما إن نقوم بكشف منطقة مُستهدفة على الخريطة. حينما لا تتبقّى أية منطقة مستهدفة، فهذا يعني أن كل الصناديق متواجدة فوق هذه المناطق (لم تتبقّ سوى صناديق `BOX_OK`).

يكفي أن نختبر ما إن كان المتغير المنطقي يحمل القيمة "خطأ"، أي أنه لم تتبقّ أية منطقة مستهدفة.
في هذه الحالة، نُعطي القيمة 0 للمتغير `cont` من أجل إيقاف الجولة :

```
1 // If there’s no remaining goal, we win
2 if (!remainingGoals)
3         cont = 0;
```

اللاعب

لم يتبقّ سوى تسوية اللاعب :

```
1 // We place the player in the right position
2 position.x = playerPosition.x ▫ BLOCK_SIZE;
3 position.y = playerPosition.y ▫ BLOCK_SIZE;
4 SDL_BlitSurface(currentMario, NULL, screen, &position);
```

نحسُب وضعيته (بالبيكسل هذه المرة) و ذلك بالقيام بعملية ضرب بين `playerPosition` و `BLOCK_SIZE`.
بعد ذلك، نقوم بلصق اللاعب في الوضعية المناسبة.

القلب !

لقد قُمنا بكلّ شيء، يكفي أن نُظهر الشاشة للمستعمل :

```
1 SDL_Flip(screen);
```

نهاية الدالة : إلغاء التحميل

بعد الحلقة الرئيسية، يجدر بنا القيام بتحرير الذاكرة التي حجزناها للـsprites التي حمّلناها.
نقوم أيضا بتعطيل تكرار الضغط على الأزرار و ذلك بإعطاء القيمة 0 للدالة `SDL_EnableKeyRepeat` :

```
// Disabling keys repetition (reset to 0)
SDL_EnableKeyRepeat(0, 0);
// Freeing the used surfaces
SDL_FreeSurface(wall);
SDL_FreeSurface(box);
SDL_FreeSurface(boxOK);
SDL_FreeSurface(level);
for (i = 0 ; i < 4 ; i++)
        SDL_FreeSurface(mario[i]);
```

الدالة movePlayer

هذه الدالة متواجدة أيضاً في الملف `game.c`. هي دالة ... معقّدة جدّا من ناحية كتابتها. و ربّما هي الدالة الأكثر صعوبة حينما نريد برمجة لعبة Sokoban.

تذكير : الدالة `movePlayer` تختبر ما إن كان لدينا الحق في تحريك اللاعب في الإتجاه المطلوب. تقوم بتحديث وضعية اللاعب `playerPosition` و أيضاً بتحديث الخريطة إذا تم تحريك صندوق.

هذا نموذج الدالة :

```
void movePlayer(int map[][NB_BLOCKS_HEIGHT], SDL_Rect *pos,int direction);
```

هذا النموذج خاص قليلاً. تلاحظ أنني أبعث الجدول `map` و أحدد الحجم الخاص بالبُعد الثاني (`NB_BLOCKS_HEIGHT`).
لماذا هذا ؟

الإجابة معقّدة قليلاً لكي أقوم بمناقشتها في وسط هذا الفصل. لكي نبسّط الأمور، لغة الـC لا تتكهّن بأننا نتحدّث عن جدول ثنائي الأبعاد و أنه يجب أن نعطي على الأقل حجم البُعد الثاني لكي تشتغل الأمور.
إذا، حينما تبعث جدولاً ذا بُعدين إلى دالة، يجب أن تحدّد حجم البُعد الثاني للجدول في النموذج. هكذا تعمل الأمور، إن الأمر ضروري.

أمر آخر : تلاحظ أن `playerPosition` تُسمى `pos` في هذه الدالة. لقد اخترت اختصار الاسم لكي تسهل كتابته بما أننا سنحتاج إلى كتابته عدّة مرات، لكي لا نتعب.

فلنبدأ باختبار الإتجاه الذي نريد التوجّه إليه و ذلك باستعمال `switch` ضخم :

```
switch(direction)
{
        case UP:
        /* etc */
```

فلننطلق في رحلة من الاختبارات المجنونة !

يجب الآن أن نكتب الاختبارات الخاصة بكلّ حالة ممكنة محاولين ألا ننسى أية واحدة.

هكذا تقوم الخطة التي أعتمدها : أختبر كل الحالات الممكنة للاصطدامات حالة بحالة، و ما إن أكشف عن اصطدام (أي أن اللاعب غير متمكّن من التحرّك) أضع الأمر `break` لأخرج من الـ`switch`، و بهذا أمنع التحرّك.

هذا مثال عن كلّ حالات الاصطدام المتواجدة للاعب يريد التحرّك نحو الأعلى :

- اللاعب متواجد أصلاً في أقصى أعلى الخريطة.
- يوجد جدار فوق اللاعب.
- يوجد صندوقان معاً فوق اللاعب (و هو غير قادر على دفع صندوقين).
- يوجد صندوق فوق اللاعب و الصندوق متواجد في الحافة العلوية للخريطة.

إذا مرّت كلّ هذه الاختبارات، يمكننا تحريك اللاعب.

سأريك الاختبارات اللازمة من أجل التحرّك نحو الأعلى. من أجل الحالات الأخرى، يكفي تعديل الشفرة قليلا.

```
1 if (pos->y - 1 < 0) // If the player exceeds the screen, we stop
2         break;
```

نبدأ بالتحقق ما إن كان اللاعب متواجداً أعلى النافذة. بالفعل، لو نحاول أن نطلب الخانة `map[5][-1]` مثلاً، سيتوقف البرنامج بشكل خاطئ !
نبدأ إذا بالتأكد من أننا لن "نتجاوز" الشاشة.

بعد ذلك :

```
1 if (map[pos->x][pos->y - 1] == WALL) // If there's a wall, we stop
2         break;
```

هنا أيضاً، الأمر بسيط. نتحقق من عدم وجود جدار فوق اللاعب. إذا كان هناك واحد، نتوقّف (`break`).

بعد ذلك (حافظ على عينيك) :

```
1 // If we want to push a box, we have to verify that there's no wall behind it (
    or another box, or the world's limit)
2 if ((map[pos->x][pos->y - 1] == BOX || map[pos->x][pos->y -1] == BOX_OK) && (
    pos->y - 2 < 0 || map[pos->x][pos->y - 2] == WALL || map[pos->x][pos->y -
    2] == BOX || map[pos->x][pos->y - 2] == BOX_OK))
3         break;
```

هذا الاختبار الضخم يمكن ترجمته كالتالي : "إذا كان هناك صندوق فوق اللاعب (أو صندوق في الوضعية المناسبة) و إذا كان فوق هذا الصندوق يوجد إما الفراغ (سنتجاوز من الحافّة العلوية لأننا في أقصى الأعلى)، أو صندوق آخر، أو صندوق في الوضعيّة المناسبة : إذاً لا يمكننا التحرّك : خروج (`break`)".

إذا تمكّنا من عبور هذا الإختبار فنحن قادرون على التحرّك، أووووف !
نستدعي اولاً دالة تقوم بتحريك الصندوق إن كنا بحاجة إلى ذلك :

```
// If we are here, so we can move the player
// We verify if there's a box to move first
moveBox(&map[pos->x][pos->y - 1], &map[pos->x][pos->y - 2]);
```

تحريك الصناديق : moveBox

قررت معالجة تحرّك الصناديق باستعمال دالة أخرى لأن الشفرة تبقى نفسها من أجل الإتجاهات الأربعة. يجب فقط أن تتأكّد بأننا قادرون على التحرّك (هذا ما كنتُ بصدد شرحه).
سنبعث للدالة معاملين : محتوى الخانة التي نريد الذهاب إليها و محتوى الخانة التي تليها.

```
void moveBox(int *firstSquare, int *secondSquare)
{
        if (*firstSquare == BOX || *firstSquare == BOX_OK)
        {
                if (*secondSquare == GOAL)
                        *secondSquare = BOX_OK;
                else
                        *secondSquare = BOX;
                if (*firstSquare == BOX_OK)
                        *firstSquare = GOAL;
                else
                        *firstSquare = EMPTY;
        }
}
```

هذه الدالة تقوم بتحديث الخريطة و هي تأخذ كمعاملات مؤشّرات نحو الخانات المعنيّة.
سأتركك لتقرأها، فهي سهلة للفهم. لا يجب أن ننسى أننا إذا حرّكنا `BOX_OK`. يجب تعويض المكان الذي كان به بـ `OBJECTIVE`. و إلا، إذا كان `BOX`، سنعوّض مكانه بـ `EMPTY`.

تحريك اللاعب

نعود للدالة `movePlayer`.
نحن هنا في الحالة الصحيحة، سنقوم بتحريك اللاعب.

كيف نفعل ذلك ؟ هذا أمر سهل :

```
pos->y--; // Finally, we can move up the player (ouf!)
```

يكفي أن ننقِص من الترتيبة (لأن اللاعب يريد الصعود للأعلى).

تلخيص

كملخّص، هاهي كلّ الاختبارات اللازمة من أجل الصعود إلى الأعلى :

```
switch(direction)
{
        case UP:
        if (pos->y - 1 < 0) // If the player exceeds the screen, we stop
                break;
        if (map[pos->x][pos->y - 1] == WALL) // If there's a wall, we stop
                break;
        // If we want to push a box, we have to verify that there's no wall
           behind it (or another box, or the world's limit)
        if ((map[pos->x][pos->y - 1] == BOX || map[pos->x][pos->y - 1] ==
           BOX_OK) && (pos->y - 2 < 0 || map[pos->x][pos->y - 2] == WALL ||
           map[pos->x][pos->y - 2] == BOX || map[pos->x][pos->y - 2] == BOX_OK
           ))
                break;
        // If we are here, so we can move the player
        // We verify if there's a box to move first
        moveBox(&map[pos->x][pos->y - 1], &map[pos->x][pos->y - 2]);
        pos->y--; // Finally, we can move up the player (ouf!)
        break;
```

سأترك لك عناء نقل الشفرة و تعديلها من أجل الحالات الأخرى (احذر، عليك ملاءمة الشفرة، ليست مطابقة تماما في كلّ مرّة !).

ها قد انتهينا من كتابة شفرة اللعبة !
حسناً، قريباً : بقي لنا أن نرى دالة التحميل و حِفظ المستويات. سنرى بعد ذلك كيف نقوم بكتابة شفرة مُنشئ المستويات.
كن متأكّدا، سيكون هذا سريعا !

4.24 تحميل و حِفظ المستويات

الملف `files.c` يحتوي على دالتين :

- `loadLevel`.
- `saveLevel`.

فلنبدأ بتحميل المستوى.

تحميل المستوى loadLevel

هذه الدالة تأخذ معاملا : الخريطة. هنا أيضاً، يجب تحديد مقدار البُعد الثاني للجدول لأننا نتكلم عن جدول ذو بعدين.
الدالة تُرجع متغيرا منطقيا : ”صحيح” إذا تم التحميل بنجاح، ”خطأ” إذا فشل.

النموذج إذا هو :

```c
int loadLevel(int level[][NB_BLOCKS_HEIGHT]);
```

فلنرى بداية الدالة :

```c
FILE* file = NULL;
char fileLine[NB_BLOCKS_WIDTH * NB_BLOCKS_HEIGHT + 1] = {0};
int i = 0, j = 0;
file = fopen(”levels.lvl”, ”r”);
if (file == NULL)
        return 0;
```

نقوم بإنشاء جدول للتخزين المؤقّت للنتيجة الخاصة بتحميل المستوى.
نفتح الملف بوضع ”قراءة فقط” (`r`). نوقف الدالة و ذلك بإرجاع القيمة 0 (”خطأ”) إذا فشلت عملية فتح الملف.
عملية تقليديّة.

الملف `levels.lvl` يحتوي على سطر و الذي هو عبارة عن تتالي أرقام. كل رقم يمثّل خانة من المستوى، مثلا :

```
11111001111111111400000111110001100103310101101100000200121110 [...]
```

يمكننا إذا قراءة هذا السطر باستعمال `fgets` :

```c
fgets(fileLine, NB_BLOCKS_WIDTH * NB_BLOCKS_HEIGHT + 1, file);
```

سنقوم بتحليل محتوى `fileLine` . نحن نعرف أن أول 12 محرفا تمثل السطر الأول، الـ12 محرفا الموالية تمثل السطر الموالي، إلى آخره.

```c
for (i = 0 ; i < NB_BLOCKS_WIDTH ; i++)
{
        for (j = 0 ; j < NB_BLOCKS_HEIGHT ; j++)
        {
                switch (fileLine[(i * NB_BLOCKS_WIDTH) + j])
                {
                        case ’0’:
                        level[j][i] = 0;
                        break;
                        case ’1’:
                        level[j][i] = 1;
                        break;
                        case ’2’:
                        level[j][i] = 2;
```

```
15                     break;
16                     case '3':
17                     level[j][i] = 3;
18                     break;
19                     case '4':
20                     level[j][i] = 4;
21                     break;
22             }
23     }
24 }
```

بواسطة عملية حسابية بسيطة، نأخذ الحرف الذي يهمّنا في `fileLine` و نحلل قيمته.

> !
> إنها "حروف" مخزّنة في الملف. ما أريد أن أقوله بهذا هو أن `'0'` مخزّن كحرف `'0'` ASCII و أن قيمته ليست 0 !
> لنحلل الملف، يجب الاختبار بـ`case '0'` و ليس `case 0` ! احذر من الخلط بين الحروف و الأرقام !

يقوم الـ`switch` بالتحويل : `'0'` ⇐ `0`، `'1'` ⇐ `1`، إلخ. يقوم بوضع الكلّ في الجدول `map`. الخريطة تسمّى `level` في هذه الدالة لكن هذا لا يغيّر أي شيء.

ما إن يتم هذا، يمكننا غلق الملف و إرجاع القيمة 1 لنقول أن كلّ شيء تمّ على ما يُرام.

```
1 fclose(file);
2 return 1;
```

أخيراً، تحميل المستوى من الملف لم يكن معقداً. الفخّ الوحيد الذي وُجب تجنّبه هو التفكير في تحويل القيمة ASCII `'0'` إلى الرقم `0` (نفس الشيء بالنسبة لـ`1`، `2`، `3`، `4`، ...).

حِفظ المستوى saveLevel

هذه الدالة أسهل :

```
1  int saveLevel(int level[][NB_BLOCKS_HEIGHT])
2  {
3          FILE* file = NULL;
4          int i = 0, j = 0;
5          file = fopen("levels.lvl", "w");
6          if (file == NULL)
7          return 0;
8          for (i = 0 ; i < NB_BLOCKS_WIDTH ; i++)
9          {
10                 for (j = 0 ; j < NB_BLOCKS_HEIGHT ; j++)
11                 {
12                         fprintf(file, "%d", level[j][i]);
```

```
13                }
14        }
15        fclose(file);
16        return 1;
17 }
```

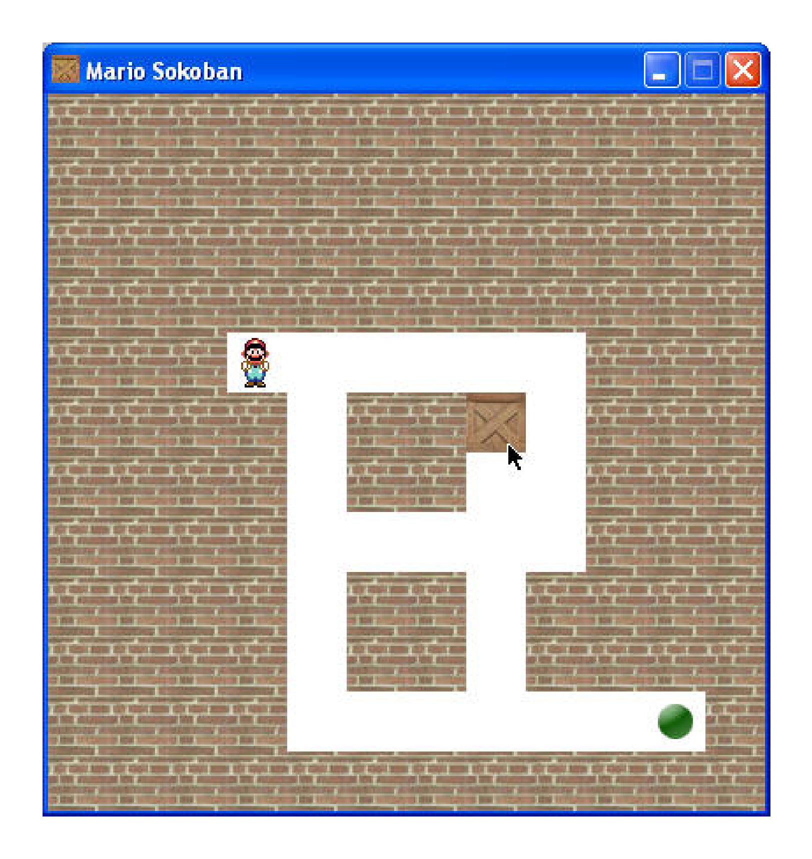
استعملت الدالة `fprintf` من أجل "ترجمة" أعداد الجدول إلى حروف ASCII. كانت هنا الصعوبة الوحيدة : تجب عدم كتابة `0` و إنما `'0'`.

5.24 مُنشئ المستويات

هذا الأخير سهل الكتابة أكثر مما تخيل.
بالمناسبة، هذه تقنية تسمح بزيادة عمر لعبتنا، فلما نتجاهلها ؟

هكذا تسري الأمور :

- نستعمل الفأرة لوضع الكتل التي نريدها في النافذة.
- النقر باليمين يسمح بمسح الكتلة الذي نتواجد فوقها الفأرة.
- النقر باليسار يسمح بوضع شيء على الخريطة. هذا الشيء يكون مخزّناً : افتراضيّا، نقوم بوضع الجدران بالنقر الأيسر للفأرة. يمكننا تغيير الشيء الذي نريد وضعه في الخريطة بالضغط على الأزرار المتواجدة في لوحة الأرقام :
  1. جدار.
  2. صندوق.
  3. منطقة مُستهدفة.
  4. مكان انطلاق Mario.
- بالضغط على `S` يتم حفظ المستوى.
- يمكننا الرجوع إلى القائمة الرئيسية بالضغط على `Esc`.



التهيئات

بشكل عام، تشبه هذه الدالّة، الدالة الخاصة باللعبة. ولذلك فقط بدأت في كتابتها باستعمال "نسخ-لصق" لدالة اللعبة، و بعد ذلك قمتُ بنزع ما لا أحتاجه و أضفت ميزات جديدة.

هذه كانت البداية :

```
1  void editor(SDL_Surface* screen)
2  {
3          SDL_Surface *wall = NULL, *box = NULL, *level = NULL, *mario = NULL;
4          SDL_Rect position;
5          SDL_Event event;
6          int cont = 1, leftClickInProgress = 0, rightClickInProgress = 0;
7          int currentObject = WALL, i = 0, j = 0;
8          int map[NB_BLOCKS_WIDTH][NB_BLOCKS_HEIGHT] = {0};
9          // Loading the objects and the level
10         wall = IMG_Load("wall.jpg");
11         box = IMG_Load("box.jpg");
12         level = IMG_Load("level.png");
13         mario = IMG_Load("mario_bas.gif");
14         if (!loadLevel(map))
15                 exit(EXIT_FAILURE);
```

هنا تجد تعريف المتغيرات و التهيئات اللازمة.

تلاحظ أنني لا أقوم بتحميل إلا Mario واحد (المتّجه نحو الأسفل). في الواقع، لن نقوم بتوجيه Mario بلوحة المفاتيح و إنما نحتاج إلى ملصق يمثّل وضعية الانطلاق الخاصة به.

المتغير `currentObject` يحفظ الشيء الذي يختاره المُستعمل حالياً. افتراضيّاً، هذا الشيء هو `WALL`. أي أننا في البداية إذا نقرنا بالزرّ اليسار سنقوم بوضع جدار، لكن يمكن تغيير هذا بواسطة المستعمل و ذلك بالضغط على `1`، `2`، `3` أو `4`.

المتغيرات المنطقية `leftClickInProgress` و `rightClickInProgress` كما تشير أسماؤها، تسمح بحفظ ما إن كان هناك نقر باليمين حالياً (أي أن زر الفأرة مضغوط). سأشرح لك المبدأ لاحقاً. على أي حال، هذه التقنية تسمح لنا بإضافة أشياء إلى الخريطة بترك زر الفأرة مضغوطاً، و إلا فسنكون مجبرين على الضغط على الزر عدة مرات من أجل وضع نفس الشيء عدّة مرات في الخريطة في أمكنة مختلفة، و هذا أمر متعب قليلا.

أخيراً، يتم تحميل الخريطة المحفوظة حالياً في الملف `levels.lvl`. سيكون نقطة انطلاقنا.

معالجة الأحداث

هذه المرة سيكون علينا معالجة كثير من الأحداث المختلفة. هيا بنا، واحداً واحداً.

SDL_QUIT

```
case SDL_QUIT:
cont = 0;
break;
```

إذا ضغطنا على الزر `X`، تتوقف الحلقة و نعود إلى القائمة الرئيسية.
ليكن في علمك أن هذا الشيء ليس أحسن حلّ بالنسبة للاعب : فهو يريد الخروج من اللعبة و ليس الرجوع إلى القائمة الرئيسية. يجب أن نجد حلاً لإيقاف البرنامج و ذلك بإرجاع قيمة خاصّة للدالة الرئيسية مثلاً. سأتركك لتجد حلاً بنفسك.

SDL_MOUSEBUTTONDOWN

```
case SDL_MOUSEBUTTONDOWN:
if (event.button.button == SDL_BUTTON_LEFT)
{
        // We put the chosen object (wall, box) in the click position
        map[event.button.x / BLOCK_SIZE][event.button.y / BLOCK_SIZE] =
            currentObject;
        leftClickInProgress = 1; // We put in mind that there’s a pushed button
}
else if (event.button.button == SDL_BUTTON_RIGHT) // Right click to erase
{
        map[event.button.x / BLOCK_SIZE][event.button.y /BLOCK_SIZE] = EMPTY;
        rightClickInProgress = 1;
}
break;
```

نبدأ باختبار الزر المضغوط (نرى ما إن كان ضغطاً بالزر الأيسر أو الأيمن) :

- إذا كان ضغطا بالزر الأيسر، نقوم بوضع الشيء الحالي `currentObject` على الخريطة في الموضع الذي تشير إليه الفأرة.
- إذا كان ضغطا بالزر الأيمن، نمسح مايوجد في الموضع الحالي للفأرة (نضع `EMPTY` كما سبق و قلتُ لك).

؟

كيف نعرف في أي "خانة" من الخريطة نحن متواجدون ؟

نعرف ذلك عن طريق عملية حسابية صغيرة. يكفي أن نأخذ إحداثيات الفأرة (`event.button.x` مثلاً) و نقسم هذه القيمة على حجم كتلة `BLOCK_SIZE`.

هذه قسمة لأعداد صحيحة. و بما أن قسمة الأعداد الصحيحة في لغة C تُعطي عدداً صحيحا، فنتحصّل بالتأكيد على قيمة توافق خانة من الخريطة.

مثلاً، لو أنني في البيكسل الـ75 من الخريطة (على محور الفواصل x)، أقسم هذا العدد على `BLOCK_SIZE` و التي تساوي هنا 34. يكون لدينا هنا :

$$75/34 = 2$$

. لا تنس هنا أننا نتجاهل باقي القسمة و نقوم بحفظ الجزء الصحيح فقط لأننا نتكلم عن قسمة أعداد صحيحة.

نحن نعلم إذا أننا نتواجد في الخانة رقم 2 (أي الخانة الثالثة لأن الجدول يبدأ من 0، لا تنس ذلك).

مثال آخر : لو أنني في البيكسل العاشر (أي أنني قريب من الحافة)، ستكون لدينا العملية الحسابية التالية :

$$10/34 = 0$$

أي أننا في الخانة رقم 0 !

بفضل هذه العملية الحسابية البسيطة يمكننا أن نعرف في أي خانة من الخريطة نحن متواجدون.

```
1 map[event.button.x / BLOCK_SIZE][event.button.y / BLOCK_SIZE] = currentObject;
```

شيء آخر مهم : إعطاء القيمة 1 للمتغير المنطقي `leftClickInProgress` (أو `rightClickInProgress` حسب الحالة) يسمح لنا بمعرفة، خلال حدث `MOUSEMOTION`، ما إن كان زر الفأرة مضغوطاً خلال الإنتقال.

SDL_MOUSEBUTTONUP

```
1 case SDL_MOUSEBUTTONUP: // We disable the boolean which indicates that there's
     a clicked button
2 if (event.button.button == SDL_BUTTON_LEFT)
3         leftClickInProgress = 0;
4 else if (event.button.button == SDL_BUTTON_RIGHT)
5         rightClickInProgress = 0;
6 break;
```

الحدث `MOUSEBUTTONUP` يقوم ببساطة بإعادة القيمة 0 للمتغير المنطقي. نحن نعرف بأن النقر انتهى و بهذا لا يوجد أي "نقر حالي" بالفأرة.

SDL_MOUSEMOTION

```
case SDL_MOUSEMOTION:
if (leftClickInProgress) // If we move the mouse and the left button is clicked
{
        map[event.motion.x / BLOCK_SIZE][event.motion.y /BLOCK_SIZE] =
            currentObject;
}
else if (rightClickInProgress) // The same thing for the right button
{
        map[event.motion.x / BLOCK_SIZE][event.motion.y / BLOCK_SIZE] = EMPTY;
}
break;
```

هنا يمكن لنا رؤية أهمية المتغيرات المنطقية. نختبر حينما نقوم بتحريك الفأرة ما إن كان هناك نقر حالي. إذا كانت هذه هي الحالة، نضع على الخريطة شيئاً ما (أو الفراغ إذا كان نقرا باليمين).
هذا يسمحُ لنا بوضع شيء واحد لعدة مرات دون الحاجة إلى النقر في كلّ مرة من أجل كلّ تكرار للشيء، يكفي إذاً أن نُبقي زر الفأرة مضغوطاً بينما نسحبُ هذه الأخيرة.

الأمر واضح : في كلّ مرة نحرّك فيها الفأرة (يكون ذلك ببيكسل واحد)، نختبر ما إن كانت المتغيرات المنطقية مفعّلة. إذا كان الأمر كذلك، نقوم بوضع شيء على الخريطة. و إلاًّ، لا نقوم بأي شيء.

ملخّص : سألخّص التقنية لأنها ستكون مفيدة من أجل برامج أخرى.
تسمح هذه التقنية بمعرفة ما إن كان زر الفأرة مضغوطاً بينما يتم تحريك هذه الأخيرة. يمكننا أن نستفيد من هذا الأمر لبرمجة السحب و الإفلات (Drag and drop).

1. خلال حدث `MOUSEBUTTONDOWN` : نعطي القيمة 1 للمتغير المنطقي `clickInProgress`.
2. خلال حدث `MOUSEMOTION` : نختبر ما إن كان المتغير المنطقي `clickInProgress` يساوي "صحيح". إذا كان الأمر كذلك فسنعرف أننا نقوم بالسحب باستخدام الفأرة.
3. خلال حدث `MOUSEBUTTONUP` : نعيد القيمة 0 للمتغير المنطقي `clickInProgress` لأن النقر قد انتهى (إفلات زر الفأرة).

SDL_KEYDOWN

تسمح أزرار لوحة المفاتيح بتحميل و حفظ المستوى و أيضاً بتغيير الشيء المُختار من أجل النقر اليساري بالفأرة.

```
case SDL_KEYDOWN:
switch(event.key.keysym.sym)
{
        case SDLK_ESCAPE:
```

```
                cont = 0;
                break;
        case SDLK_s:
                saveLevel(map);
                break;
        case SDLK_c:
                loadLevel(map);
                break;
        case SDLK_KP1:
                currentObject = WALL;
                break;
        case SDLK_KP2:
                currentObject = BOX;
                break;
        case SDLK_KP3:
                currentObject = GOAL;
                break;
        case SDLK_KP4:
                currentObject = MARIO;
                break;
}
break;
```

هذه الشفرة سهلة للغاية. نقوم بتغيير الشيء إذا تم الضغط على الأرقام في اللوحة، نقوم بحفظ المستوى إذا تم الضغط على `S` و نقوم بتحميل آخر مستوى تم حفظه بالنقر على `C`.

وقت اللصق !

ها نحن ذا : لقد أتممنا كلّ الأحداث.
الآن، لم يتبقّ لنا سوى لصق كل عناصر الخريطة بمساعدة حلقتين متداخلتين. الشفرة التالية تشبه الشفرة التي استعملناها في دالة اللعبة. سأعطيها لك لكنّي لن أعيد شرحها هنا :

```
// Clearing the screen
SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 255, 255, 255));
// Placing the objects in the screen
for (i = 0 ; i < NB_BLOCKS_WIDTH ; i++)
{
        for (j = 0 ; j < NB_BLOCKS_HEIGHT ; j++)
        {
                position.x = i * BLOCK_SIZE;
                position.y = j * BLOCK_SIZE;
                switch(map[i][j])
                {
                        case WALL:
                        SDL_BlitSurface(wall, NULL, screen, &position);
                        break;
                        case BOX:
```

```
                        SDL_BlitSurface(box, NULL, screen, &position);
                        break;
                        case GOAL:
                        SDL_BlitSurface(level, NULL, screen, &position);
                        break;
                        case MARIO:
                        SDL_BlitSurface(mario, NULL, screen, &position);
                        break;
                }
        }
}
// Updating the screen
SDL_Flip(screen);
```

لا يجب أن ننسى أن نحرر الذاكرة بعد الانتهاء من الحلقة الرئيسية بالشكل اللازم (باستعمال `SDL_FreeSurface`) :

```
SDL_FreeSurface(wall);
SDL_FreeSurface(box);
SDL_FreeSurface(level);
SDL_FreeSurface(mario);
```

حسناً، انتهينا من التنظيف !

ملخّص و تحسينات

حسناً لقد انتهينا من كلّ شيء و حان وقت التلخيص !

هيّا فلنلخّص !

و ماذا سيكون أحسن تلخيص من الشفرة المصدرية الكاملة للعبة مع التعليقات المفصّلة ؟

بسبب عدم رغبتي في كتابة عشرات الصفحات من الشفرة تشمل كلّ ما رأيناه إلى حدّ الآن، أفضّل أن تقوم بتنزيل الشفرة المصدريّة الكاملة مع الملف التنفيذي (المُترجم للويندوز).

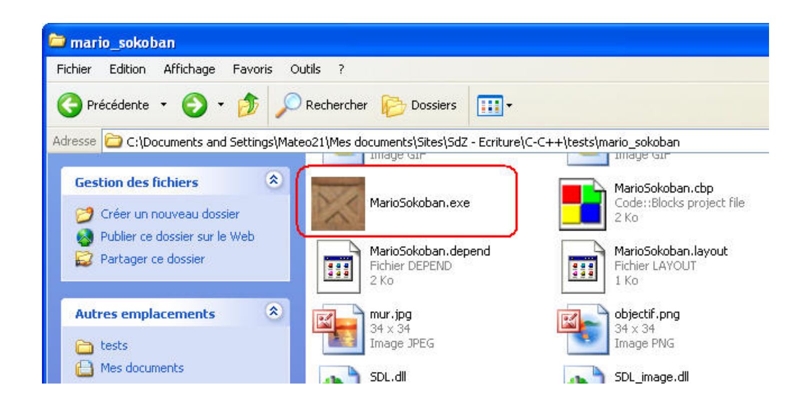
https://openclassrooms.com/uploads/fr/ftp/mateo21/mario_sokoban.zip (436 Ko)

الملف `.zip` يحتوي :

- الملف التنفيذي للويندوز (إذا كنت تعمل على نظام تشغيل آخر، تكفي إعادة الترجمة).
- الملفات DLL الخاصة بالـSDL و الـSDL_Image.
- كل الصور التي تحتاجها في البرنامج (هي نفسها التي قمت بتحميلها في حزمة "sprites" أعلاه).
- الملفات المصدرية الكاملة الخاصة بالبرنامج.

- الملف `.cbp` الخاص بمشروع Code::Blocks. إذا أردت فتح المشروع باستعمال بيئة تطويرية أخرى، قم بإنشاء مشروع SDL، أضف إليه يدوياً كل الملفات `.h` و `.c` . الأمر ليس صعباً، سترى.

تلاحظ أن المشروع يحوي، بالإضافة إلى الملفات `.h` و `.c`، ملفا مصدريا `ressources.rc` . إنه ملف يمكن إضافته للمشروع (فقط على الويندوز) و يسمح بإدخال ملفات في الملف التنفيذي. هنا، استعنت به لإدخال أيقونة في الملف التنفيذي. و هذا يسمح بإعطاء أيقونة للملف التنفيذي مرئية في الويندوز، أنظر الصورة التالية :



إِعترف بذلك، صنع أيقونة من أجل البرنامج أمر أجمل من ترك الأيقونة الافتراضيّة !
يمكنك قراءة المزيد عن هذه التقنيّة في درس "إنشاء أيقونة لبرنامجك" المتوفّر على هذا الرابط :

http://www.siteduzero.com/tutoriel-3-14177-creer-une-icone-pour-son-programme.html

حسّن اللعبة !

ألا ترى بأن هذا البرنامج غير مثالي، و أبعد من أي يكون كذلك ؟
هل تريد أفكاراً للتطوير ؟

- ينقص **دليل استعمال**، حيث يتم إظهار شاشة قبل انطلاق المرحلة و قبل انطلاق مُنشئ المستويات. نقوم بشرح الأزرار اللازمة لكي يستعملها اللاعب.

- في مُنشئ المستويات، اللاعب لا يعرف أيّ شيء مُختار حالياً. سيكون من الجيّد أن **الشيء المختار حاليّا** يتبع مؤشّر الفأرة. هكذا سيعرف المستخدم مالّذي سيوضع على الخريطة. الأمر سهل للتطبيق، لقد قمنا بجعل Zozor يتبع الفأرة في الفصل السابق !

- يمكن أن نبدأ مستوى ما بوجود بعض **الصناديق الموضوعة أساساً فوق المناطق المستهدفة** (`BOX_OK`). لقد رأيت كثيرا من المستويات تبدأ بصناديق في مكان مناسب هذا (لا يعني أن المستوى سهل، فقد يكون عليك تحريك الصندوق من مكانه في مرحلة من مراحل الجولة).

- **في مُنشئ المستويات أيضاً يجب أن نمنع المستعمل من أن يضع موضِعي انطلاق للاعب في نفس الخريطة !**

- حينما نجح في مُستوى، نرجع مباشرة إلى القائمة الرئيسية. هذا أمر فضّ نوعاً ما، ما رأيك **بإظهار رسالة** في وسط الشاشة : "هنيئاً، لقد نجحت في المستوى" ؟
- أخيراً، سيكون من الجيد أن يتمكن البرنامج من التحكم في عدة مستويات في المرة الواحدة. سيكون علينا بناء **رحلة لعب تستمر لـ20 مستوى مثلاً**. سيكون الأمر أصعب قليلاً من ناحية البرمجة، لكن يمكن القيام به. يجب عليك التعديل في شفرة اللعبة و أيضاً في شفرة مُنشئ المستويات. أنصحك بأن تضع مستوى واحداً في السطر الواحد بالملف `levels.lvl`.

كما وعدتك، هذا ممكن، و لقد فعلته ! لن أعطيك الشفرة المصدرية الخاصة بهذه التحسينات (أعتقد أنّي أعطيتك الكثير إلى حدّ الآن !)، و لكنّي سأعطيك مباشرة الملف التنفيذي مُترجماً للويندوز و اللينكس.

اللعبة تحتوي على مغامرة من 20 مستوى تختلف صعوبتها (من سهل جداً إلى ... شديد الصعوبة).
لكي أتمكّن من تحقيق بعض المستويات، احتجت لزيارة موقع شخص مهووس بلعبة Sokoban :

http://sokoban.online.fr/

هاهي اللعبة المحسّنة للويندوز و اللينكس :

Windows : https://openclassrooms.com/uploads/fr/ftp/mateo21/mario_sokoban_setup.exe (656 Ko)

Linux : https://openclassrooms.com/uploads/fr/ftp/mateo21/mario_sokoban_linux.tar.gz (64 Ko)

لقد استعنت بالبرنامج Inno Setup من أجل صنع برنامج التسطيب، يمكنك القراءة بهذا الخصوص على هذا الرابط :

http://www.siteduzero.com/tutoriel-3-14171-creer-une-installation.html

الفصل 25

تحكّم في الوقت !

لهذا الفصل أهمية كبيرة : سيعلّمك كيف تحكّم في الوقت بالـSDL. إنه لمن النادر أن نقوم بإنشاء برنامج SDL لا تحتاج إلى دوال خاصة بالتحكّم في الوقت، بالرغم من أن لعبة Mario Sokoban كانت حالة خاصة. رغم ذلك، في معظم الألعاب، إدارة الوقت هي شيء أساسيّ.

مثلاً، كيف لك أن تُنشئ لعبة Tetris أو Snake ؟ يجب فعلاً على الكُتل أن تحرّك كل X ثانية، و هذا ما لا تجيد فعله. على الأقل، قبل أن تقرأ هذا الفصل.

1.25 الـDelay و Ticks

في بادئ الأمر، ستتعلّم كيف نستعمل دالتين بسيطتين جداً :

- `SDL_Delay` : تسمح بتوقيف البرنامج مؤقتا لعدد من الميلّي ثواني.
- `SDL_GetTicks` : تقوم بإرجاع عدد الميلّي ثواني التي مضت منذ انطلاق تشغيل البرنامج.

هاتان الدالّتان سهلتان جداً كما سنرى لكنّ استعمالهما ليس بسيطاً كما يبدو الأمر عليه.

SDL_Delay

كما قلتُ، تقوم هذه الدالة بإيقاف عمل البرنامج لمدّة محدّدة. حينما يكون البرنامج متوقّفا، نقول أنّه "ينام" (sleep) : هو لا يستعمل المُعالج.

يمكن إذا استعمال `SDL_Delay` للإنقاص من زمن اشتغال المُعالج (Processor). لاحظ أنّني سأختصره إلى CPU و هذا الإختصار متداول و يوافق العبارة "*Central Processing Unit*" و الّتي تعني "وحدة المعالجة المركزيّة".
بفضل الـ`SDL_Delay` يمكنك جعل برامجك أقلّ شراهة لموارد المعالج. أي أننا لن نثقل على الحاسوب كثيراً إذا تمّ استخدام هذه الدالة بذكاء.

> م
> هذا كلّه يعتمد على البرنامج الذي تنُشئه : أحياناً، نجد أنه من المستحسن أن يستعمل البرنامج المعالج بشكل أقلّ، يمكن في نفس الوقت أن يقوم المُستعمل بشيء آخر مثلما هو الحال بالنسبة لقارئ MP3 الذي يشتغل في الخلفية ريثما تقوم بالتصفّح عبر الإنترنت.
> لكن أحياناً، نحتاج للبرنامج أن يستعمل المعالج بنسبة 100%، و هو الحال بالنسبة لغالبية الألعاب.

لنعد إلى الدالة، هذا نموذجها و هو بسيط للغاية :

```c
void SDL_Delay(Uint32 ms);
```

الأمر واضح، تبعث للدالة عدد الميلّي ثواني التي يجب أن "ينام" البرنامج خلالها.

مثلاً : إذا أردت أن ينام البرنامج لمدّة ثانية واحدة، يجب عليك كتابة :

```c
SDL_Delay(1000);
```

لا تنس أنّها بالميلي ثانية :

- 1000 ميلّي ثانية = ثانية.
- 500 ميلّي ثانية = نصف ثانية.
- 250 ميلّي ثانية = رُبع ثانية.

> !
> لا يمكنك فعل أي شيء في البرنامج بينما هو متوقّف مؤقّتا ! فالبرنامج "النائم" لا يمكن له فعل أي شيء لأنه ليس مفعّلا بالنسبة للحاسوب.

مشكل جزئيّة الوقت

لا، تأكّد بأنني لن أخوض في درس للفيزياء الكميّة في هذا الفصل حول الـSDL ! و مع ذلك، أرى بأن هناك أمورا يجب عليك معرفتها : `SDL_Delay` ليست دالة "مثالية". و هذا ليس خطأها، بل هو خطأ نظام التشغيل (Windows، GNU/Linux، Mac OS X ...).

لماذا يتدخّل نظام التشغيل هنا ؟ ببساطة لأنه هو الذي يتحكّم في البرامج المشغّلة ! فبرنامجك سيقول للنظام : "سأنام، أيقظني بعد ثانية". لكن لن يقوم النظام دائما بإفاقة البرنامج بعد ثانية بالضبط.

في الواقع، قد يكون هناك تأخّر بسيط (تأخر 10 ميلّي ثانية بالتقريب كمعدّل، هذا يختلف حسب الحاسوب). لماذا ؟ لأن CPU لا يمكنه العمل إلا على برنامج واحد في المرّة الواحدة. دور نظام التشغيل يتمثّل في إخبار CPU بخصوص ما يجب أن

يتم القيام به و لهذا : "لمدّة 40 ميلّي ثانية ستعملُ على `firefox.exe` ثم لمدّة 110 ميلّي ثانية على `explorer.exe`، بعد ذلك، لمدة 80 ميلّي ثانية ستعمل على `program_sdl.exe` ثم عد إلى العمل على `firefox.exe` لمدّة 65 ميلّي ثانية ... " نظام التشغيل هو بالفعل عبارة عن قائد الأوركسترا !

تخيّل الآن أنه لثانية من الزمن، يكون برنامج آخر لازال في طور الإشتغال : يجب أن ينتهي عمله حتى يستطيع برنامجك "استعادة التحكّم" على CPU.

ما الذي يجب تذكّره ؟ أن CPU لا يمكنه أن يتحكّم في برنامجين في آن واحد. و لكي يعطي انطباعاً بأنه يُشغّل العديد من البرامج في نفس اللحظة، يقوم بتقسيم الوقت بين هذه البرامج حيث تعملُ دوراً بدور.
قلَّت صحّة هذا الكلام لأن المعالجات "ثنائية النوى" لها القدرة على تشغيل برنامجين في نفس الآن.

لكن هذه التقنية في التحكّم بالبرامج معقّدة للغاية و لن نحصل على ضمانات بأنّ البرنامج الخاص بنا سيتمّ إيقاظه في ثانية بالضبط من الزمن.

مع ذلك، يعتمد الأمر دائماً على الحاسوب نفسه كما قلتُ سابقاً. عندي، ألاحظ أنّ الدالّة `SDL_Delay` دقيقة جدّاً.

> بسبب مشكل جزئيّة الوقت، لن تمكّن إذا من إيقاف برنامجك مؤقّتا لوقت قصير جدّاً من الزمن، أي أنه لو استعملت `SDL_Delay(1);` ستكون متأكداً بأن البرنامج لن ينام لـ1 ميلي ثانية و إنما أكثر (حوالي 9 أو 10 ميلي ثانية).

الدالّة `SDL_Delay` عمليّة، لكن لا تثق بها كثيراً. فهي لا توقف البرنامج بالمقدار الزمني الذي تحدده أنت بالضبط.
هذا ليس راجعاً لكون الدالة غير مُبرمجة جيداً، لكن لأن عمل الجهاز معقّد و لا يمكنه أن يكون دقيقاً من هذه الناحية.

SDL_GetTicks

هذه الدالة تُرجع عدد الميلي ثواني التي انقضت منذ بدأ عمل البرنامج. و هي عبارة عن مؤشّر للزمن لا يمكن الاستغناء عنه. ستجد بأنها مفيدة لكي تضع مراجع في الزمن، سترى ذلك !

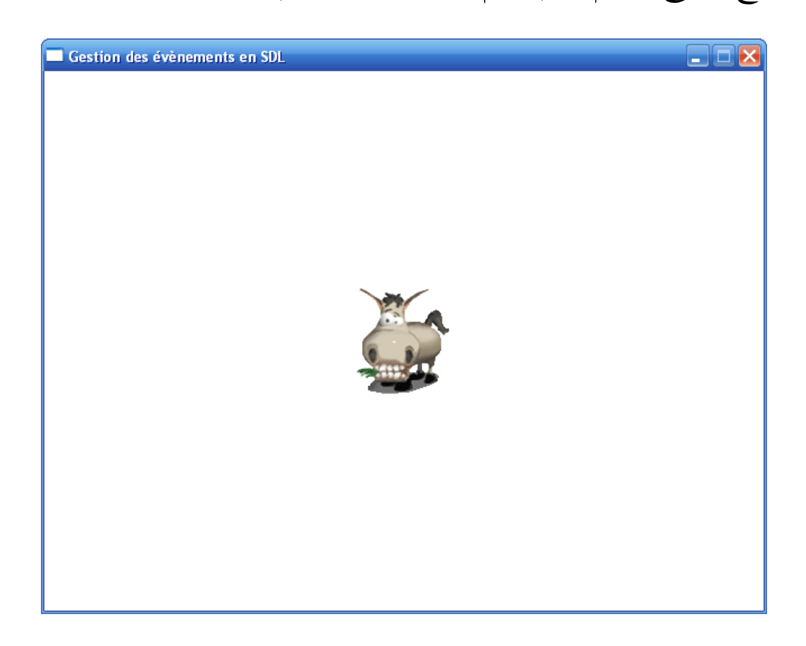
هذا نموذجها :

```c
Uint32 SDL_GetTicks(void);
```

هذه الدالة لا تنتظر أي معامل، و هي تقوم فقط بإرجاع عدد الثواني المنقضية.
هذا العدد يتصاعد مع مضي الزمن. لمعلوماتك، التوثيق الخاص بالـSDL يشير إلى أن هذا العدد يصل إلى الحد الأقصى 49 يوما ثم يبدأ العدد من جديد ! لكن يجدر بالبرنامج الذي تكتبه ألا يستمرّ كلّ هذا الوقت و لهذا فلا تقلق من هذه الناحية.

إستعمال SDL_GetTicks لإدارة الوقت

إذا كانت الدالة `SDL_Delay` سهلة للفهم و للاسعمال، فالأمر ليس عينه بالنسبة لـ `SDL_GetTicks` . حان الوقت لنعرف كيف سنستفيد منها.
إليك هذا المثال، سنسترجع البرنامج القديم الذي يقوم بإظهار نافذة تحتوي على Zozor (الصورة التالية) :



هذه المرة، في عوض التحكّم في حركة Zozor بالفأرة أو بلوحة المفاتيح، سنعتمد فكرة أنه سيقوم بالتحرّك لوحده في الشاشة. لنبسّط الأمور، سنجعله يتحرّك أفقيا في النافذة.

سنُعيد استعمال نفس الشفرة التي استخدمناها في فصل الأحداث، يجدر بك أن تجيد كتابتها بنفسك دون الحاجة لُمساعدة مني. و إلا، إذا احتجتها، يمكنك استعادتها من الفصول السابقة.

```c
int main(int argc, char *argv[])
{
        SDL_Surface *screen = NULL, *zozor = NULL;
        SDL_Rect zozorPosition;
        SDL_Event event;
        int cont = 1;
        SDL_Init(SDL_INIT_VIDEO);
        screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
        SDL_WM_SetCaption("Gestion du temps en SDL", NULL);
        zozor = SDL_LoadBMP("zozor.bmp");
        SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0,
            255));
        zozorPosition.x = screen->w / 2 - zozor->w / 2;
        zozorPosition.y = screen->h / 2 - zozor->h / 2;
        SDL_EnableKeyRepeat(10, 10);
        while (cont)
        {
                SDL_WaitEvent(&event);
                switch(event.type)
```

```
19                {
20                        case SDL_QUIT:
21                        cont = 0;
22                        break;
23                }
24                SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 255, 255,
                      255));
25                SDL_BlitSurface(zozor, NULL, screen, &zozorPosition);
26                SDL_Flip(screen);
27        }
28        SDL_FreeSurface(zozor);
29        SDL_Quit();
30        return EXIT_SUCCESS;
31 }
```

فلنهتم بـZozor. نريد أن نحرّكه. من أجل هذا، سيكون من الأفضل استعمال `SDL_GetTicks`. سنحتاج إلى متغيرين : `previousTime` و `currentTime`. يستخدمان من أجل تخزين الوقت الذي تمّ إرجاعه من طرف `SDL_GetTicks` في لحظات زمنية مختلفة.
يكفي أن نحسب الفرق بين `currentTime` و `previousTime` لمعرفة الوقت المنقضّي. إذا كان هذا الأخير يساوي 30 ميلي ثانية، نغيّر إحداثيات Zozor.

لنبدأ إذا بإنشاء هذين المتغيرين :

```
1 int previousTime = 0, currentTime = 0;
```

و الآن، في حلقتنا غير المنتهية، نضيف الشفرة المصدرية التالية :

```
1 currentTime = SDL_GetTicks();
2 if (currentTime - previousTime > 30) // If 30 ms have passed
3 {
4         zozorPosition.x++; // We move Zozor
5         previousTime = currentTime; // The current time becomes the previous
              one.
6 }
```

إفهم جيّدا ما يحصل :

1. نحصل على الوقت المنقضي باستعمال `SDL_GetTicks`.
2. نقارن هذه القيمة بالوقت الذي تم تسجيله مسبقاً. إذا كان هناك فرق 30 مث على الأقل، إذا،..
3. نحرّك Zozor، لأننا نريده أن يتحرّك كلّ 30 مث. هنا، نقوم بتحريكه إلى اليمين كلّ 30 مث.
يجب ان نتأكد ما إن كان الوقت المنقضي أكبر من 30 مث، و ليس ما إن كان يساوي تلك القيمة ! لأنه في الواقع سنُخْبَر ما إن كان الوقت المنقضي يساوي على الأقل 30 مث. نحن لسنا متأكدين بأنه سيتمّ تنفيذ الأمر كلّ 30 مث بالضبط.

4. ثمّ، و الأمر الذي لا يجب فعلاً نسيانه، نضع قيمة الوقت "الحالي" في الوقت "السابق". بالفعل، تخيّل الدورة القادمة للحلقة : الوقت الحالي يتغيّر و يمكننا مقارنته بالوقت السابق من جديد، أي سنقارن ما إن تم انقضاء 30 مث على الأقل ثم نحرّك Zozor.

> ؟ و لكن ماذا يحصل لو أن الحلقة اشتغلت لمدّة أقل من 30 مث ؟

إقرأ جيداً الشفرة، لا شيء سيحدث !

لن ندخل في الشرط، يعني أننا لن نقوم بأي شيء. ننتظر الدورة القادمة للحلقة أين نقوم من جديد باختبار ما إن تم انقضاء 30 مث منذ آخر مرّة قمنا فيها بتحريك Zozor.

هذه الشفرة قصيرة، لكن يجب فهمها ! أعد قراءة شرحي بالعدد اللازم من المرّات لتفهم جيداً لأن هذا الجزء قد يكون الأهم في هذا الفصل.

تغيير في معالجة بالأحداث

الشفرة المصدرية الذي كتبناها مثالية إلى حدّ ما إذ ينقصها تفصيل بسيط : الدالة `SDL_WaitEvent`. كانت هذه الدالة عمليّة إلى حدّ الآن بما أننا لم نتحكّم في الوقت. هذه الدالة توقف البرنامج مؤقّتا (بنفس طريقة `SDL_Delay` تقريباً) ما دام لا يوجد أي حدث.

لكن هنا، لسنا مُضطرين إلى انتظار حدث لنقوم بتحريك Zozor ! إذ يجب عليه التحرّك لوحده.
و لا يجب عليك الاستمرار في تحريك الفأرة فقط لإنتاج أحداث و منه الخروج من الدالة `SDL_WaitEvent` !

ماهو الحلّ ؟ `SDL_PollEvent`.
لقد قدّمت لك من قبل هذه الدالة : على عكس `SDL_WaitEvent`، تُرجع هذه الدالة قيمة سواء كان هناك حدث أم لا. و نقول بأن الدالة غير معُطّلِة : هي لا توقف البرنامج مؤقّتا لأن الحلقة غير المنتهية ستستمرّ في العمل طوال الوقت.

الشفرة المصدرية الكاملة

هذه هي الشفرة النهائية التي بإمكانك تجريبها :

```
int main(int argc, char *argv[])
{
        SDL_Surface *screen = NULL, *zozor = NULL;
        SDL_Rect zozorPosition;
        SDL_Event event;
        int cont = 1;
        int previousTime = 0, currentTime = 0;
        SDL_Init(SDL_INIT_VIDEO);
        screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
```

```
10	SDL_WM_SetCaption("Gestion du temps en SDL", NULL);
11	zozor = SDL_LoadBMP("zozor.bmp");
12	SDL_SetColorKey(zozor,SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0,
	   255));
13	zozorPosition.x = screen->w / 2 - zozor->w / 2;
14	zozorPosition.y = screen->h / 2 - zozor->h / 2;
15	SDL_EnableKeyRepeat(10, 10);
16	while (cont)
17	{
18		SDL_PollEvent(&event); // We use PollEvent and not WaitEvent in
		   order not to block the program
19		switch(event.type)
20		{
21			case SDL_QUIT:
22			cont = 0;
23			break;
24		}
25		currentTime = SDL_GetTicks();
26		if (currentTime - previousTime > 30) // If 30ms have passed
		   since the final loop iteration
27		{
28			zozorPosition.x++; // We move Zozor
29			previousTime = currentTime; // The current time becomes
			   the previous one for our future calculation.
30		}
31		SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 255, 255,
		   255));
32		SDL_BlitSurface(zozor, NULL, screen, &zozorPosition);
33		SDL_Flip(screen);
34	}
35	SDL_FreeSurface(zozor);
36	SDL_Quit();
37	return EXIT_SUCCESS;
38 }
```

يجدُر بك أن ترى Zozor يهتزّ لوحده على الشاشة. هو يتحرّك نحو اليمين.
حاول مثلاً تغيير الوقت من 30 مث إلى 15 مث : يجدر بـZozor أن يتحرّك إلى اليمين بشكل أسرع بمرتين ! في الواقع، هو يتحرّك مرّة كل 15 مث في عوض مرّة كل 30 مث كالسابق.

استهلاك أقل للـCPU

حالياً، البرنامج يدور في حلقة غير منتهية بسرعة الضوء (حسنا، تقريباً). و لهذا فهو يستهلك 100% من المعالج. لكي نرى هذا يكفي مثلاً أن نضغط على CTRL + ALT + DEL (في القائمة Processes) في Windows :

ElkCtrl.exe 00 5 120 Ko
testsdl.exe 100 6 444 Ko
CameraAssistant.exe 00 7 324 Ko
LVCOMSX.EXE 00 5 420 Ko
wmplayer.exe 756 Ko
HydraDM.exe 032 Ko
100% du CPU

كما يمكنك أن ترى، يتم استعمال CPU بنسبة 100% من طرف برنامجنا `testsdl.exe`.
لقد قلت لك مسبّقاً : إذا برمجت لعبة (خاصة إذا كانت بنظام شاشة كاملة)، ليس خطيراً أن تستعمل المعالج بنسبة 100%. لكن إذا كانت لعبة في نافذة مثلاً، يُستحسن استعمال نسبة أقل من CPU لكي نسمح للمستعمل بالقيام بشيء آخر دون أن يُجهد الحاسوب نفسه.

الحل ؟ سنقوم بإعادة الشفرة السابقة، لكننا سنضيف إليه `SDL_Delay` من أجل انتظار الوقت اللازم لكي يصل إلى 30 مث.

يكفي أن نضيف `SDL_Delay` في `else` :

```
currentTime = SDL_GetTicks();
if (currentTime - previousTime > 30) // If 30ms have passed
{
        zozorPosition.x++; // We move Zozor
        previousTime = currentTime; // The current time becomes the previous
            one for our future calculation
}
else
{
        SDL_Delay(30 - (currentTime - previousTime));
}
```

كيف تعمل الأمور هذه المرة ؟ الأمر بسيط، هناك احتمالان (حسب الشرط) :

- إما أنه مضت اكثر من 30 مث منذ قمنا بتحريك Zozor، في هذه الحالة نحرّكه.
- إما أنه مضى وقت أقل من 30 مث، في هذه الحالة سينام البرنامج بفضل `SDL_Delay` ريثما يسمح بوصول الـ30 مث و بهذه العملية الحسابية التي قُمنا بها `30 - (currentTime - previousTime)`. إذا كان الفرق بين الزمن الحالي و الزمن السابق هي 20 مث مثلاً، فسينام البرنامج لـ 30 - 20 = 10 مث لكي تصل الـ30 مث.

> تذكّر بأن `SDL_Delay` يمكن لها أن تضيف بعض الميلّي ثواني أكثر من المتوقع.

بهذه الشفرة، سينام البرنامج معظم الوقت و بهذا نقلل من استهلاك CPU. لاحظ الصورة التالية :

ElkCtrl.exe 00 5 120 Ko
testsdl.exe 00 6 444 Ko
CameraAssistant.exe 00 7 324 Ko
LVCOMSX.EXE 00 5 420 Ko
wmplayer.exe 756 Ko
HydraDM.exe 032 Ko
0% du CPU

يستعمل البرنامج كمعدّل حوالي 0 إلى 1% من CPU... أحياناً يستعمل أكثر بقليل لكنّه يعود سريعاً إلى 0%.

التحكّم في عدد الصور في الثانية

أنت تتساءل حتماً كيف يمكننا الحدّ من (أو تثبيت) عدد الصور في الثانية (غالباً ما نسمّي هذا FPS اِختصاراً لFrames per second) التي يُظهرها الحاسوب.

حسناً، هذا تماماً ما نحاول فعله ! فهنا نقوم بإظهار صورة جديدة كل 30 مث كمعدّل. علماً أن ثانية واحدة تساوي 1000 مث، لكي نجد عدد الFPS، يجب أن نقوم بعملية قسمة 1000 / 30 = 33 صورة في الثانية بالتقريب.

بالنسبة لعين الإنسان، نقول عن التحرّك أنه رشيق إذا احتوى على الأقل 25 صورة في الثانية. بـ33 صورة في الثانية إذاً فالتحرّك في البرنامج رشيق تماما و بهذا لن يظهر متشنّجا.

إذا أردنا صورا أكثر في الثانية، يجب إنقاص حدود الوقت بين صورتين. انتقل من 30 إلى 20 مث و ستصبح العملية : 1000 / 20 = 50 FPS.

تمارين

التحكّم في الوقت ليس أمراً بديهياً، سيكون من الجيد لك أن تتمرّن، ما رأيك ؟ إليك بعض التمارين :

- لحدّ الآن، يتحرّك Zozor في كلّ مرة إلى اليمين إلى أن يختفي من الشاشة. سيكون من الأفضل حينما يصل إلى حافة النافذة أن يعيد التوجّه إلى اليسار. سيكون ذلك أفضل أليس كذلك ؟ لأنه سيعطي انطباعاً أنه يرتدّ.

  أنصحك بإنشاء متغير منطقي `toTheRight` يحمل القيمة "صحيح" إذا كان Zozor يتحرّك نحو اليمين (و "خطأ" إذا كان يتحرّك نحو اليسار). إذا كان المتغير المنطقي يحمل القيمة صحيح، تقوم بتحريك Zozor إلى اليمين، و إلا فستقوم بتحريكه إلى اليسار. لا تنس أن تغيّر قيمة المتغير المنطقي ما إن يصل Zozor إلى حافة النافذة و ذلك لكي ينطلق في الإتجاه المُعاكس !

- بدل أن يقوم Zozor بالإرتداد من اليمين إلى اليسار، فيمكن تحريكه على قطر النافذة ! يكفيك تغيير `zozorPosition.x` و `zozorPosition.y` في نفس الوقت. يمكنك رؤية ماذا يُعطينا الأمر لو نقوم بزيادة قيمة `x` و إنقاص قيمة `y` في نفس الوقت، أو إذا قُمنا بزيادة قيمتيهما معاً، إلخ.

- حاول جعل Zozor يتوقّف عن التحرّك إذا تمّ الضغط على الزر `P`، و إذا تم الضغط مجدداً على نفس الزر ينطلق Zozor مجدداً. يحتاج الأمر متغيرا منطقيا بسيطا تقوم بتفعيله أو تعطيله.

2.25 المؤقِتات (Timers)

> ! استعمال المؤقِتات أكثر تعقيداً قليلاً لأننا سنعتمد على مبدأ لم نره لحدّ الآن : المؤشّرات نحو الدوال. استعمال المؤقِتات ليس أمراً ضروريا : إذا وجدت بأنها صعبة جداً لتستعملها، يمكنك غضّ النظر عنها دون أي مشكل.

المؤقِتات تشكّل طريقة أخرى لتحقيق ما نحن بصدد رؤيته بالدالة `SDL_GetTicks`.
هي تقنية خاصّة نوعاً ما. بعض المبرمجين يجدونها عمليّة، و آخرون لا. هذا يعتمد على ذوقك البرمجي.

ماهو المؤقِت ؟
هو نظام يسمح بالطلب من الـSDL أن تستدعي دالة ما كل X ميلّي ثانية. يمكنك بهذا أن تنشئ دالة `moveEnemy()` تقوم الـSDL باستدعائها تلقائيّا كل 50 مث كي يستطيع العدوّ التحرّك في مجالات معيّنة.

> م كما كنت أقول لك الآن، يمكننا القيام بهذا بواسطة `SDL_GetTicks` باستعمال التقنية التي رأيناها أعلاه.
> ما الفائدة إذا ؟ لنقل أن المؤقِتات تفرض علينا هيكلة برامجنا بشكل أفضل على شكل دوال.

تهيئة نظام المؤقِتات

لكي تتمكّن من استعمال المؤقِتات، يجب علينا تهيئة الـSDL أولاً باستعمال عَلَم خاص : `SDL_INIT_TIMER`. يجب عليك إذا استدعاء الدالة `SDL_Init` كالتالي :

```c
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER);
```

الـSDL جاهزة الآن لاستعمال المؤقِتات !

إضافة مؤقِت

لنضيف مؤقِتاً يجب علينا استدعاء الدالة `SDL_AddTimer`. إليك نموذجها :

```c
SDL_TimerID SDL_AddTimer(Uint32 interval, SDL_NewTimerCallback callback, void
    param);
```

توجد في الواقع دالتان تسمحان بإضافة مؤقِت في الـSDL هما : `SDL_AddTimer` و `SDL_SetTimer` و هما تقريباً متطابقتان. و مع ذلك، `SDL_SetTimer` هي دالة قديمة موجودة دائما لأسباب التوافقيّة (Compatibility). حاليّا، إذا أردنا القيام بالأمر بالشكل الجيد، أنصحك باستعمال `SDL_AddTimer`.

الدالة تستقبل ثلاثة معاملات :

- **المجال الزمني** (بالميلي ثانية) بين كلّ استدعاء للدالة و آخر.
- **اسم الدالة التي نريد استدعاءها.** نسمّي هذه بدالة الرد (Callback) : يتكفّل البرنامج باستدعاء الدالة بشكل دوري.
- **المعاملات** التي نبعثها لدالة الرد.

؟

كيف يمكن لاسم دالة أن يكون معاملا ؟ اعتقدتُ أن المعاملات لا يمكنها أن تكون إلا أسماء متغيرات !

في الواقع، يتم أيضاً حْفظ الدوال في الذاكرة خلال تحميل البرنامج. فهي تملك أيضاً عناوين خاصة بها. لهذا، يمكننا أن ننُشئ مؤشّرات نحو دوال ! تكفي كتابة اسم الدالة التي نريد استدعاءها للإشارة إلى عنوانها. و بهذا، ستعرف الـSDL العنوان بالذاكرة الذي يجب أن تذهب إليه لاستدعاء دالة الرد.
إذا أردت معرفة المزيد حول المؤشرات نحو الدوال، أدعوك إلى قراءة الدرس التعليمي المكتوب بواسطة العضو mleg (في الموقع الفرنسي) و الذي يحلّل هذا الموضوع :

http://www.siteduzero.com/tutoriel-3-314203-les-pointeurs-sur-fonctions.html

تُرجع الدالة `SDL_AddTimer` عدداً خاصاً بالمؤقِت (ID). يجدر بك تخزين هذه النتيجة في متغير من نوع `SDL_TimerID`. هذا سيسمح لك لاحقاً بتعطيل المؤقِت : يكفي أن تشير إلى ID المؤقت و سيتمّ إيقافه.

تسمح لنا الـSDL بتفعيل الكثير من المؤقِتات في نفس الوقت. هذا يشرح الفائدة من تخزين هويّة كلّ مؤقِت لكي نفرق بينها.

سنُنشئ إذا هوية المؤقِت :

```
1 SDL_TimerID timer; // A variable to save the number of timer
```

ثم نقوم بإنشاء المؤقِت :

```
1 timer = SDL_AddTimer(30, moveZozor, &zozorPosition); // Starting the timer
```

هنا، أنشئ مؤقِتا يحمل المعاملات التالية :

- يتم استدعاؤه كلّ 30 مث.
- يقوم باستدعاء دالة الرد المسمّاة `moveZozor`.
- يبعث له كمعامل، مؤشّراً نحو وضعية Zozor لكي يتمكّن من التعديل عليه.

لقد فهمت المبدأ : دور الدالة `moveZozor` هو تغيير وضعية Zozor كل 30 مث.

إنشاء دالة الرد

احذر : يجب أن تكون حذراً هنا. يجب أن يكون نموذج دالة الرد هو التالي إجباريا :

```
1 Uint32 functionName(Uint32 interval, void *parameter);
```

لكي ننشئ دالة الرد المسمّاة `moveZozor` ، يجب أن نكتب الدالة كالتالي :

```
1 Uint32 moveZozor(Uint32 interval, void *parameter);
```

إليك الشفرة الخاصة بالدالة `moveZozor` ، إنها معقّدة أكثر مما تبدو عليه :

```
1 // Callback function (Will be called every 30 ms)
2 Uint32 moveZozor(Uint32 interval, void *parameter)
3 {
4         SDL_Rect* zozorPosition = parameter; // Automatic conversion from void*
              to SDL_Rect*
5         zozorPosition->x++;
6         return interval;
7 }
```

يتم استدعاء الدالة `moveZozor` تلقائيًا كل 30 مث بواسطة الـSDL. تقوم هذه الأخيرة ببعث معاملين تماماً للدالة (لا أكثر و لا أقل) :

- **المجال الزمني** الذي يفرّق كل استدعائين للدالة (هنا 30 مث).
- **المعامل "المخصّص"** الذي طلبت إعطاءه للدالة. لاحظ، و من المهم جداً، أن هذا المعامل هو عبارة عن مؤشّر نحو `void` . هذا يعني أنه مؤشّر يؤشّر نحو أي نوع كان : على `int` ، هيكل مخصّص، أو، مثل هنا، على `SDL_Rect` (`zozorPosition`).
  لاحظ أيضاً أنه لا يمكن بعث أكثر من معامل مخصّص لدالة الرد. لحسن الحظ، نحن دائمًا قادرون على إنشاء أنواع خاصة بنا (أو جداول) و التي ستكون عبارة عن تجميع لمتغيرات نريد بعثها للدالة.

المشكل هو أن هذا المعامل هو مؤشّر من نوع غير معروف (`void`) للدالة. يجب إذا أن نقول للحاسوب أن هذا المعامل هو `SDL_Rect*` (مؤشّر نحو `SDL_Rect`).

لفعل ذلك، أنشئ مؤشّرا نحو `SDL_Rect` في دالتي التي تأخذ كمعامل المؤشّر `parameter` .

؟ ما الفائدة من إنشاء مؤشّر ثانٍ ليحمل نفس العنوان ؟

الفائدة هي أن `zozorPosition` من نوع `SDL_Rect*` بعكس المتغير `parameter` الذي كان من نوع `void*` .

يمكننا إذا الوصول إلى `zozorPosition->x` و `zozorPosition->y`.
لو قمت بكتابة `parametre->x` أو `parametre->y` فالمترجم كان سيرفضها لأنّ متغيّرا من نوع `void` لا يملك هذه المركّبات.

بعد ذلك، السطر التالي بسيط : نعدّل قيمة `zozorPosition->x` لتحريك Zozor نحو اليمين.

آخر شيء (مهم جداً) : يجب عليك إرجاع المتغير `interval`. هذا يشير للـSDL بأننا نريد أن نستمر في اعتبار أنه سيتم استدعاء الدالة كل 30 مث.
إذا كنت تريد تغيير المجال الزمني لاستدعاء الدالة، يكفي أن تبعث قيمة أخرى (في غالب الأحيان لا نفعل ذلك).

إيقاف المؤقِت

لإيقاف المؤقِت، الأمر بسيط :

```c
SDL_RemoveTimer(timer); // Stopping the timer
```

يكفي إذا استدعاء `SDL_RemoveTimer` و ذلك بالإشارة إلى هوية المؤقِت الذي نريد إيقافه.
هنا أوقف المؤقِت مباشرة بعد الحلقة غير المنتهية، في نفس موضع `SDL_FreeSurface`.

ملخّص

- تمكّن الدالة `SDL_Delay` من إيقاف البرنامج مؤقّتا لعدد معين من الميلّي ثواني. هذا يسمح بإنقاص نسبة استعمال المُعالج الذي لن يكون "خلال نوم البرنامج" مستعملاً من طرف هذا الأخير.
- يمكننا معرفة عدد الميلّي ثواني المنقضية منذ اشتغال البرنامج باستعمال `SDL_GetTicks`، بواسطة عمليات حسابية بسيطة، يمكننا الاستفادة من هذا لكي نقوم بمعالجة غير معطّلة للأحداث بواسطة `SDL_PollEvent`.
- المؤقِتات تشكّل نظاما يسمح باستدعاء دوالك (المسمّاة بدوال الرد) على مجالات زمنيّة محدّدة. يمكننا التحصل على نفس النتيجة باستعمال `SDL_GetTicks` لكن المؤقِتات تساعد على هيكلة البرنامج بشكل أفضل و جعله أحسن من ناحية القراءة.

الفصل 26

كتابة نصوص باستعمال SDL_ttf

يمكنني التكهّن بأن معظم القرّاء قد طرح هذا السؤال من قبل : "و لكن، ألا توجد أي دالة لكي تكتب نصاً على نافذة SDL ؟" حان الوقت لأجيبك : الجواب هو لا.

رغم ذلك، توجد طرق لفعل هذا. يمكننا فقط ... وضع صور للحروف بجانب بعضها البعض على الشاشة. هذا الأمر يعمل لكنّه ليس عمليا.

لحسن الحظ، يوجد ماهو أبسط : يمكننا استعمال المكتبة SDL_ttf. إنها مكتبة تم إضافتها إلى الـSDL تماماً مثل الـSDL_image. دورها هو إنشاء مساحة `SDL_Surface` إنطلاقا من النص الذي نبعثه لها.

1.26 تسطيب SDL_ttf

يجب أن تعرف أنه، مثل SDL_image، SDL_ttf هي مكتبة تحتاج إلى أن تكون المكتبة SDL مثبّتة من قبل. حسناً : إذا كنت إلى حدّ الآن لم تمكّن من تسطيب المكتبة SDL فهذا أمر شنيع و لهذا فسأعتبر أنك قمت بذلك !

تماما مثل SDL_image، فإن المكتبة SDL_ttf هي واحدة من المكتبات المُرتبطة بالـSDL الأكثر شعبية (أي أنه يتم تنزيلها بكثرة). كما ستُلاحظ، هذه المكتبة مُبرمجة بشكل جيد. ما إن تجيد استعمالها لن يمكنك أن نتوقّف عن ذلك !

كيف تعمل SDL_ttf ؟

SDL_ttf لا تقوم بإظهار صور bitmap لتولّد نصا في مساحات. في الحقيقة، هي طريقة ثقيلة لفعلها و لن يتاح لنا استعمال سوى خط واحد.
في الواقع، تستدعي المكتبة SDL_ttf مكتبةً أخرى : FreeType. هي مكتبة قادرة على قراءة ملفات خطوط بصيغة `.ttf` لتُخرج منها صورة. تقوم SDL_ttf باسترجاع هذه الصورة و تحوّلها للـSDL و ذلك بإنشاء مساحة `SDL_Surface`.

و بهذا فإن SDL_ttf تحتاج المكتبة FreeType لكي تشتغل، و إلا فلن تكون قادرة على قراءة ملفات الخطوط `.ttf`.

إذا كنت تعمل بـ**Windows** و تستعمل، مثلما أفعل، النسخة المُترجمَة للمكتبة، لن تحتاج إلى تنزيل أي شيء لأن FreeType مضمّنة من قبل في المكتبة الحيّة `SDL_ttf.dll` و لهذا فليس عليك القيام بأي شيء.

إذا كنت تعمل بالـ**GNU/Linux** أو **Mac OS X** فمن اللازم أن تعيد ترجمة المكتبة، فتلزمك FreeType لتتم الترجمة.
اذهب إذن إلى صفحة تنزيل FreeType :

http://www.freetype.org/download.html#stable

لتنزيل الملفات الخاصة بالمطورين.

تثبيت SDL_ttf

اذهب إلى صفحة تنزيل SDL_ttf :

http://www.libsdl.org/projects/SDL_ttf/

هنا، اختر الملف اللازم من القسم *"Binary"*.

> في Windows، لاحظ أنه لا يوجد سوى ملفان بصيغة `.zip` يحملان في نهاية اسميهما اللاحقتين `win32` و `VC6`. الأولى (`win32`) تحتوي الـDLL التي تحتاج إلى تقديمها مع الملف التنفيذي. يجب عليك أيضاً وضع هذه الـDLL في مجلّد المشروع لتستطيع تجريب البرنامج، طبعا.
> الثانية (`VC6`) تحتوي الملفات `.h` و الملفات `.lib` التي تحتاجها للبرمجة. يمكننا أن نفكّر من خلال الاسم أن هذه الملفّات تخص Visual C++ فقط، لكن في الحقيقة، و بشكل خاص، الملف `.lib` يعمل أيضاً مع mingw32، سيشتغل إذن في الـCode::Blocks.

الملف `.zip` يحتوي كالعادة مجلد `include` و مجلد `lib`. قم بوضع محتوى المجلد `include` في المسار `mingw32/include/SDL`، و محتوى المجلد `lib` في المسار `mingw32/lib`.

> ! يجدر بك نسخ الملف `SDL_ttf.h` في المجلد `mingw32/include/SDL` و ليس في المجلد `mingw32/include` فقط. احذر الخطأ !

تخصيص مشروع من أجل الـSDL_ttf

بقيت لنا مرحلة واحدة أخيرة : تخصيص المشروع لكي يكون قادراً على استعمال SDL_ttf بشكل جيد. يجب أن يتم التعديل على خصائص محرّر الروابط لكي يُترجم البرنامج بشكل جيد و ذلك باستعمال SDL_ttf.

لقد تعلّمت من قبل هذه العملية بالنسبة لـSDL_image، و لهذا سأسرع قليلاً.
بما أنني أعمل في الـCode::Blocks سأعطيك العملية الخاصة بهذه البيئة التطويرية. بالنسبة لباقي البيئات، فالطريقة لا تختلف كثيراً عن هذه :

- توجّه نحو القائمة `Project` / `Build Options`.

- في القسم `Linker` أنقر على الزر الصغير `Add`.
- أشر إلى المسار الذي يوجد به الملف `SDL_ttf.lib` (بالنسبة لي هو في `C:\Program Files\CodeBlocks\mingw32\lib`).
- ستظهر لك هذه الرسالة : "Keep this as a relative path ?" لا يهمّ ما تختاره لأن الأمر سيشتغل في كلتا الحالتين. أنصحك أن تجيب بالسلب لأن المشروع لن يشتغل لو وضعته في مسار آخر غير المتواجد به لو أنك أجبت بالإيجاب.
- وافق على التغييرات بالنقر على `OK`.

؟

ألا نحتاج إلى ربط المكتبة FreeType أيضاً ؟

كلا، مثلما قلتُ فـFreeType مضمّنة في الـDLL الخاصة بـSDL_ttf. لهذا فلن يكون عليك الاهتمام بها، لأن SDL_ttf تفعل ذلك الآن.

الملفات التوثيقية

و الآن بما أنك أصبحت مبرمجاً محنّكاً تقريباً، يجدر بك أن تطرح التساؤل التالي : "لكن أين هو التوثيق ؟" إن لم تطرح هذا السؤال فهذا يعني أنّك لازلت لم تصبح بعد مبرمجاً محنّكاً.

يوجد بالطبع دروس تفصّل في كيفية عمل المكتبات، مثل هذا الكتاب، و لكن :

- لن أستطيع أن أضع لك فصلاً حول كل المكتبات الموجودة (حتى لو أمضيت حياتي كلّها في ذلك، لن يكفيني الوقت !). و لهذا يجب عاجلاً أم آجلا قراءة التوثيق و يجدر بك أن تتعوّد على ذلك من الآن !
- من جهة أخرى، في غالب الأحيان تكون المكتبة معقّدة نوعاً ما و تحتوي كثيراً من الدوال. لن أتمكن من تقديم كلّ هذه الدوال في هذا الفصل لأنه سيكون بذلك طويلاً جداً !

من الواضح جداً أن التوثيق يكون كاملا و يلمّ بكل خفايا المكتبة، و لهذا أفضّل أن أعطيك من الآن رابط صفحة التوثيق الخاصة بـSDL_ttf :

http://sdl.beuc.net/sdl.wiki/SDL_ttf

التوثيق متوفّر بصيغ مختلفة : HTML على الشبكة، HTML مضغوطة، PDF، إلخ. خذ النسخة التي تناسبك.

ستجد بأن SDL_ttf مكتبة بسيطة جداً : يوجد بها قليل من الدوال (حوالي 40 - 50، نعم إنها قليلة !). يجدر بهذا أن تكون إشارة (للمبرمجين المحنّكين من ضمن القرّاء) إلى أن هذه المكتبة سهلة و ستستطيع التعامل معها سريعاً.

هيا، حان الوقت لنتعلّم كيف نستخدم SDL_ttf الآن !

2.26 تحميل SDL_ttf

التضمين

قبل كلّ شيء، يجب تضمين الملف الرأسي التالي قبل كلّ استعمال لهذه المكتبة :

```
#include <SDL/SDL_ttf.h>
```

إذا صادفت أخطاء ترجمة الآن، تأكد بأنك وضعت الملف `SDL_ttf.h` في المجلّد `mingw32/include/SDL` و ليس في `mingw32/include` فقط.

تشغيل SDL_ttf

تماما مثل الـSDL، تحتاج SDL_ttf أن تُشغّل في بداية الشفرة وتُوقّف في نهايتها.
توجد دالتان تشبهان كثيراً الدالتين الخاصتين بالـSDL :

- `TTF_Init` : تقوم ببدء تشغيل SDL_ttf.
- `TTF_Quit` : توقّف SDL_ttf.

ليس واجباً أن يتم بدء تشغيل SDL قبل SDL_ttf.

لكي تقوم ببدء تشغيل SDL_ttf (نقول أيضاً تهيئة)، يجب أن نستدعي الدالة `TTF_Init` . هذه الأخيرة لا تحتاج إلى أن تستقبل أي معامل و هي تقوم بإرجاع القيمة -1 إن حدث أي خطأ.

يمكنك البدء في تشغيل SDL_ttf ببساطة كالتالي :

```
TTF_Init();
```

إذا أردت أن تتأكد ما إن كان قد حدث خطأ أم لا، جرّب الشفرة التالية :

```
if(TTF_Init() == -1)
{
        fprintf(stderr, "Error initializing TTF_Init : %s\n", TTF_GetError());
        exit(EXIT_FAILURE);
}
```

إذا كان هناك خطأ في تشغيل SDL_ttf، سيتم إنشاء ملف `stderr.txt` (في Windows على الأقل) يحتوي على رسالة تشرح الخطأ.
للذين يطرحون السؤال : الدالة `TTF_GetError` تقوم بإرجاع آخر رسالة خطأ للـSDL_ttf، و لهذا استعملتها في الـ `fprintf` .

إيقاف SDL_ttf

لنوقّف المكتبة، نستدعي الدالة `TTF_Quit` . هي أيضاً لا تحتاج أي معامل. يمكنك استدعاؤها قبل أو بعد `SDL_Quit` هذا لا يهم :

```
1 TTF_Quit();
```

تحميل خط

حسناً كان كلّ شيء جيداً و غير معقّد، لكننا لم نستمتع بعد. لننتقل إلى الأهمّ إذا أردت ذلك : و الآن بما أنه تم تحميل SDL_ttf، يجب علينا أن نقوم بتحميل خط ما. ما إن يتم هذا الشيء، يمكننا أخيراً كتابة النص !

هنا أيضاً، توجد دالتان :

- `TTF_OpenFont` : تفتح ملف خط (`.ttf`).
- `TTF_CloseFont` : تغلق الملف المفتوح.

يجدر بالدالة `TTF_OpenFont` أن تخزّن النتيجة في متغير من نوع `TTF_Font` . لهذا يجب عليك إنشاء مؤشّر من نوع `TTF_Font` كالتالي :

```
1 TTF_Font *font = NULL;
```

يحتوي المؤشّر `font` إذا على معلومات خاصة بالخط المفتوح.

تأخذ الدالة `TTF_OpenFont` معاملين :

- اسم ملف الخط (بصيغة `.ttf`) الذي نريد فتحه. الأمثل هو وضع ملف الخط في مجلّد المشروع. مثال عن ملف : `arial.ttf` (من أجل الخط Arial).
- حجم الخط الذي نريد استعماله. يمكنك مثلا استعمال حجم 22.
  إنها نفس الحجوم التي تستعملها في برامج معالجة النصوص مثل Word.

لم يتبقّ لنا سوى إيجاد الخطوط ذات الصيغة `.ttf` . أنت تملك أصلاً العديد منها على حاسوبك، لكن يمكنك تنزيلها من الأنترنت كما سنرى الآن.

على حاسوبك

لديك أصلا خطوط على حاسوبك !
إن كنت تعمل بـWindows، ستجد الكثير من هذه الملفات في المجلّد `C:\Windows\Fonts`.
ليس عليك سوى نسخ الملف الخاص بالخط الذي يعجبك و لصقه في مجلّد المشروع.

إذا كان اسم الملف يحتوي على حروف "غريبة" كالفراغات، الحروف ذات العلامات الصوتية (accents) أو حتى الحروف الكبيرة، أنصحك بإعادة تسمية هذا الملف. و لكي نكون متيقنين من عدم وجود أيّ مشكل، لا تستعمل سوى الأحرف الصغيرة و تجنّب الفراغات.

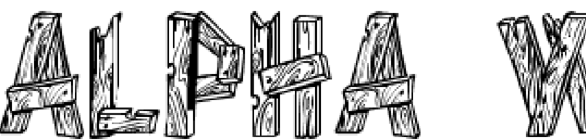
- مثال عن اسم خاطئ : `TIMES NEW ROMAN.TTF`.
- مثال عن اسم صحيح : `times.ttf`.

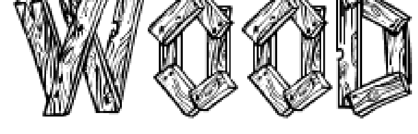
على الأنترنت

الخيار الآخر : احصل على خطّ من الأنترنت. ستجد الكثير من المواقع التي تقترح خطوطا مجانية و أصلية للتنزيل.

أنصحك شخصيا بزيارة الموقع dafont.com لأنّه مصنّف بشكل جيّد و محتواه منظّم و منوّع.

لاحظ الصور التالية التي ستعطيك فكرة عن الخطوط التي ستجدها هناك بسهولة :





RAVEN

Angelina

تحميل الخط

أقترح عليك استعمال الخط Angelina (http://www.dafont.com/angelina.font) لبقية الأمثلة.

فلنفتح الخط كالتالي :

```
font = TTF_OpenFont("angelina.ttf", 65);
```

الخط المستعمل سيكون `angelina.ttf` . لقد قمت وضع هذا الخط في مجلّد المشروع كما قمت بإعادة تسميته لكي يكون كلّه بحروف صغيرة.
سيكون للخط الحجم 65. ستبدو الكتابة كبيرة لكنه خطّ خاص يستلزم ذلك لكي يظهر بشكل جيد.

الأمر المهم هو أن `TTF_OpenFont` تخزّن النتيجة في المتغير `font` ، ستعيد استعمال هذا المتغير الآن بكتابة نص. فهي تسمح بالإشارة إلى الخط الذي نريد أن نستعمله لكي نكتب النص.

> لا تحتاج إلى فتح الخط في كلّ مرة تريد فيها الكتابة به : افتحه مرّة واحدة في بداية البرنامج و أغلقه في نهايته.

غلق الخط

يجب التفكير في غلق كل خط قمنا بفتحه قبل استدعاء `TTF_Quit` . في حالتي، هذا ما تكون عليه الشفرة :

```
1 TTF_CloseFont(font); // Must be before TTF_Quit();
2 TTF_Quit();
```

هكذا يكون العمل !

3.26 الطرق المختلفة للكتابة

و الآن، بما أنه تم تحميل SDL_ttf و أن لدينا متغيرا `font` محمّلا هو الآخر، لن يمنعنا أي شيء و أي شخص من كتابة نص في نافذة SDL !

جيد : كتابة النص هو أمر جيد، لكن بواسطة أي دالة ؟ من خلال التوثيق يوجد ما لا يقلّ عن 12 دالة لفعل ذلك !

في الواقع، توجد 3 طرق مختلفة للـSDL_ttf لكي ترسم نصاً.

- **Solid** (الصورة 1) : هي التقنية الأكثر سرعة. ستتم كتابة النص بسرعة في `SDL_Surface` . ستكون المساحة شفافة لكنها لن تستخدم إلا مستوًى واحداً من الشفافية (لقد تعلّمنا ذلك في الفصول السابقة). هذا أمر عملي، لكن النص لن يكون جميلاً لأنه حوافه لن تكون منحوتة بشكل جيد و خاصة إن كان مكتوبا بحجم ضخم. استعمل هذه التقنية حينما يكون عليك تغيير النص كثيراً، مثلا لإظهار الوقت المنقضي أو عدد الـFPS الخاص بلعبة.
- **Shaded** (الصورة 2) : هذه المرة، سيكون النص جميلاً. فالحروف ستكون محسّنة أكثر (هذا يعني أن محيط الحواف سيكون مُلطّفا بشكل مُريح لعين الإنسان) و سيظهر النص أكثر نعومة. يوجد عيب في هذه التقنية : يجب أن تكون الخلفية ذات لون واحد موحّد. يستحيل جعل خلفية الـ`SDL_Surface` شفافة بطريقة الـShaded.
- **Blended** (الصورة 3) : هي التقنية الأكثر قوّة، لكنها بطيئة. في الواقع، هي تأخذ الوقت اللازم الذي تأخذه التقنية Shaded لإنشاء الـ`SDL_Surface` . الإختلاف الوحيد بينها و بين الـShaded، هي أنه يمكنك لصق النص على صورة و سيتم احترام الشفافية (على عكس Shaded التي تفرض وجود خلفية موحّدة اللون). احذر : عملية اللصق بهذه الطريقة أبطأ من تلك الخاصة بالـShaded.





ملخّص :

- إذا كان لديك نص يتغير محتواه كثيراً، كعداد عكسي، استعمل التقنية Solid.
- إذا كان النص لا يتغير كثيراً و أنك تريد لصق النص على خلفية موحدة اللون، استعمل التقنية Shaded.
- إذا كان النص لا يتغير كثيراً و لكنك تريد لصقه على خلفية غير موحدّة اللون (كصورة مثلاً) استعمل التقنية Blended.

هكذا إذا، يجدر بك أن تكون قد تعوّدت قليلاً على هذه الأساليب الخاصة بـSDL_ttf في الكتابة.

لقد قلتُ لك أنه توجد 12 دالة لذلك.
في الواقع، من أجل كلّ طريقة في الكتابة، توجد 4 دوال لذلك. كلّ دالة تكتب النص بالاستعانة بمجموعة محارف (Charset) مختلفة. هذه الدوال هي :

- Latin1،
- UTF8،
- Unicode،
- Unicode Glyph.

الأمثل أن تختار Unicode لأنها مجموعة محارف تحوي أغلب الحروف و الإشارات الموجودة على وجه الأرض. و لكن، استعمال الـUnicode ليس سهلا دائماً (محرف واحد و هو يأخذ حجما أكبر من حجم `char` في الذاكرة)، فلن نرى كيفية استعمالها هنا.

إذا كان برنامجك مكتوباً بالفرنسية فمجموعة Latin1 تكفي بإسهاب، يمكنك الاكتفاء بهذه الأخيرة.

الدوال الثلاثة التي تستعمل نظام التشفير Latin1 هي :

- `TTF_RenderText_Solid`،
- `TF_RenderText_Shaded`،
- `TTF_RenderText_Blended`.

مثال عن كتابة نص بطريقة الـBlended

لكيّ نختار لونا بـSDL_ttf، لن نستعمل نفس النوع كما بالـSDL (إنشاء متغير من نوع `Uint32` بالاستعانة بالدالة `SDL_MapRGB`).
بالعكس، سنستعمل هيكلا جاهزا من طرف الـSDL و هو : `SDL_Color`. هذا الهيكل يحتوي ثلاثة مركّبات : كمية الأحمر، الأخضر و الأزرق.

إذا أردت إنشاء متغير `blackColor`، يجب عليك أن تكتب إذا :

```
SDL_Color blackColor = {0, 0, 0};
```

!

احذر لكي لا تخلط بينها و بين الألوان التي تستعملها عادة SDL !
الـSDL تستعمل متغيرات `Uint32` يتم إنشاؤها بمساعدة `SDL_MapRGB`.
بينما SDL_ttf تستعمل متغيرات `SDL_Color`.

سنقوم بكتابة نص بالأسود في `SDL_Surface`، نسميها `text`.

```
text = TTF_RenderText_Blended(font, "Salut les Zér0s !", blackColor);
```

أنت ترى المعاملات التي بعثناها بالترتيب : الخط (من نوع `TTF_Font`)، النص الذي نريد كتابته و أخيراً اللون (من نوع `SDL_Color`).
يتم تخزين النتيجة في مساحة. تحسب SDL_ttf تلقائياً الحجم اللازم للمساحة بدلالة حجم النص و عدد الحروف التي تريد كتابتها.

كما هو الحال بالنسبة لأي مساحة، سيحتوي المؤشّر `text` المركّبات `w` و `h` التي تشير بالترتيب إلى عرض و ارتفاع المساحة. إذن فهذه طريقة جيدة لمعرفة أبعاد النص ما إن تتم كتابة هذا الأخير على المساحة. لن يكون عليك سوى كتابة :

```
text->w // Gives the width
text->h // Gives the height
```

الشفرة المصدرية الكاملة لكتابة نص

أنت تعرف الآن كلّ ما يجب أن تتم معرفته بخصوص الـSDL_ttf، فلنرى الشفرة المصدرية التي تلخّص كتابة نص بطريقة الـBlended :

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
#include <SDL/SDL_image.h>
#include <SDL/SDL_ttf.h>
int main(int argc, char *argv[])
{
        SDL_Surface *screen = NULL, *text = NULL, *wallpaper = NULL;
        SDL_Rect position;
        SDL_Event event;
        TTF_Font *font = NULL;
        SDL_Color blackColor = {0, 0, 0};
        int cont = 1;
        SDL_Init(SDL_INIT_VIDEO);
        TTF_Init();
        screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
        SDL_WM_SetCaption("Gestion du texte avec SDL_ttf", NULL);
        wallpaper = IMG_Load("moraira.jpg");
        // Loading the font
        font = TTF_OpenFont("angelina.ttf", 65);
        // Writing the text on the surface with blended mode (the optimal one)
        text = TTF_RenderText_Blended(font, "Salut les Zér0s !", blackColor);
        while (cont)
        {
                SDL_WaitEvent(&event);
                switch(event.type)
                {
                        case SDL_QUIT:
                        cont = 0;
                        break;
                }
                SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 255, 255,
                    255));
                position.x = 0;
                position.y = 0;
                SDL_BlitSurface(wallpaper, NULL, screen, &position); //
                    Blitting the wallpaper
                position.x = 60;
                position.y = 370;
                SDL_BlitSurface(text, NULL, screen, &position); // Blitting the
                    text
                SDL_Flip(screen);
        }
        TTF_CloseFont(font);
        TTF_Quit();
```

```
43        SDL_FreeSurface(text);
44        SDL_Quit();
45        return EXIT_SUCCESS;
46 }
```

النتيجة تمثّلها الصورة التالية :



إذا أردت تغيير طريقة الكتابة للتجريب، لا يوجد سوى سطر للتعديل : السطر الخاص بإنشاء المساحة (استدعاء الدالة `TTF_RenderText_Blended`).

> ! تأخذ الدالة `TTF_RenderText_Shaded` معاملا رابعا على عكس الأخرتين. هذا المعامل الأخير هو لون الخلفية الذي نريد استعماله. يجب عليك إذا إنشاء متغير من نوع `SDL_Color` للإشارة إلى لون الخلفية (مثلا أبيض).

خصائص كتابة نص

يمكن أيضاً تحديد خصائص الخط، كغليظ مثلاً، مائل و مسطّر.

يجب أولاّ أن يتم تحميل الخط و لهذا يجب أن يتوفر لديك متغير `font` صحيح. و يمكنك إذا استدعاء الدالة `TTF_SetFontStyle` التي ستقوم بالتعديل على الخط لكي يكون غليظا، مائلا أو مسطّرا حسب الرغبة. الدالة تأخذ معاملين :

- الخط الذي نريد تعديله.
- دمج أعلام للإشارة إلى نمط الكتابة الذي نريد إعطاءه : غليظ، مائل أو مسطّر.

بالنسبة للأعلام، يجب عليك استعمال الثوابت التالية :

- `TTF_STYLE_NORMAL` : عادي.
- `TTF_STYLE_BOLD` : غليظ.
- `TTF_STYLE_ITALIC` : مائل.
- `TTF_STYLE_UNDERLINE` : مسطّر.

بما أنها قائمة من الاعلام، يمكنك الدمج بينها باستعمال الإشارة `|` كما تعلّمنا القيام بذلك سابقاً.

فلنجرّب :

```
1 // Loading the font
2 font = TTF_OpenFont("angelina.ttf", 65);
3 // The text will be italic and underlined
4 TTF_SetFontStyle(font, TTF_STYLE_ITALIC | TTF_STYLE_UNDERLINE);
5 // Writing the text in italic and underlined modes
6 text = TTF_RenderText_Blended(font, "Salut les Zér0s !", blackColor);
```

النتيجة : النص مكتوب بخاصية مائل و مسطّر :



لإرجاع خط ما إلى حالته العاديّة، يكفي أن نعيد استدعاء الدالة `TTF_SetFontStyle` باستعمال العلم `TTF_STYLE_NORMAL` هذه المرّة.

تمرين : العداد

سيجمع هذا التمرين بين المفاهيم التي تعلّمتها في هذا الفصل و فصل التحكّم في الوقت. مهمّتك، إن قبلتها، هي إنشاء عداد تتصاعد قيمته كلّ أعشار الثانية، أي أنه سيظهر بشكل تقدّمي القيم التالية : 0، 100، 200، 300، 400 ... بعد ثانية، يجدر بالرقم 1000 أن يظهر.

طريقة للكتابة في سلسلة محارف

لكي تحلّ هذا التمرين، ستحتاج إلى معرفة كيفية الكتابة داخل سلسلة محارف في الذاكرة.
في الواقع يجب عليك أن تعطي للدالة `TTF_RenderText` متغيرا من نوع `char*` لكن ماهو متوفّر لديك هو عدد (من نوع `int` مثلا). كيف يمكننا تحويل عدد إلى سلسلة محارف ؟

يمكننا أن نستعمل من أجل هذا الدالة `sprintf`.
إنها تعمل بنفس الطريقة التي تعمل بها `fprintf`، الاختلاف الوحيد هو أنه في عوض الكتابة في ملف، ستتم الكتابة في سلسلة محارف (الحرف s يختصر الكلمة string و التي تعني "سلسلة محارف" بالإنجليزيّة).
أوّل معامل تقدّمه سيكون إذا مؤشّرا نحو جدول من `char`.

> x
>
> قم بحجز مكان كافٍ من أجل جدول `char` إذا أردت ألا تتجاوز في الذاكرة !

مثال :

```
1  sprintf(time, "Temps : %d", counter);
```

هنا، المتغير `time` هو جدول محارف (20 محرفا)، و `counter` هو متغير من نوع `int` يحوي الزمن.
بعد هذه التعليمة، سلسلة المحارف `time` ستحتوي مثلا على "Temps : 500".

هيّا، حان وقتُ العمل !

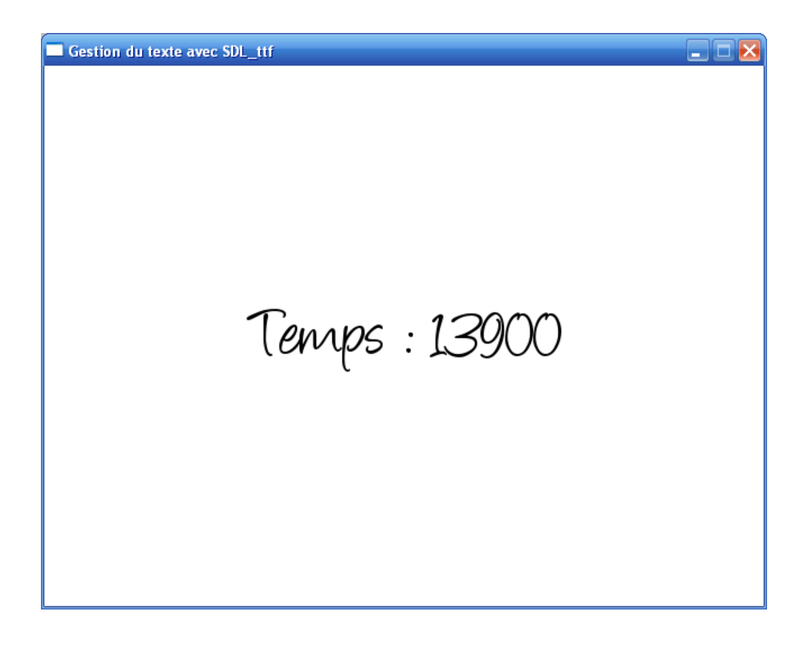
التصحيح

هذا تصحيح ممكن للتمرين :

```
1  int main(int argc, char *argv[])
2  {
3          SDL_Surface *screen = NULL, *text = NULL;
4          SDL_Rect position;
5          SDL_Event event;
6          TTF_Font *font = NULL;
7          SDL_Color blackColor = {0, 0, 0}, whiteColor = {255, 255, 255};
8          int cont = 1;
9          int currentTime = 0, previousTime = 0, counter = 0;
10         char time[20] = ""; // A table of char big enough
11         SDL_Init(SDL_INIT_VIDEO);
12         TTF_Init();
13         screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
14         SDL_WM_SetCaption("Gestion du texte avec SDL_ttf", NULL);
15         // Loading the police
16         font = TTF_OpenFont("angelina.ttf", 65);
17         // Time and text initialization
```

```
18         currentTime = SDL_GetTicks();
19         sprintf(time, "Temps : %d", counter);
20         text = TTF_RenderText_Shaded(font, time, blackColor, whiteColor);
21         while (cont)
22         {
23                 SDL_PollEvent(&event);
24                 switch(event.type)
25                 {
26                         case SDL_QUIT:
27                         cont = 0;
28                         break;
29                 }
30                 SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 255, 255,
                        255));
31                 currentTime = SDL_GetTicks();
32                 if (currentTime - previousTime >= 100) // If 100ms at least
                       have passed
33                 {
34                         counter += 100; // We add 100ms to the counter
35                         sprintf(time, "Temps : %d", counter); // We write in
                               the string "time" the new time
36                         SDL_FreeSurface(text);// We delete the previous surface
37                         text = TTF_RenderText_Shaded(font, time, blackColor,
                               whiteColor); // We write the sring "time" in
                               SDL_Surface
38                         previousTime = currentTime; // We update the previous
                               time
39                 }
40                 position.x = 180;
41                 position.y = 210;
42                 SDL_BlitSurface(text, NULL, screen, &position); // Blitting the
                        text
43                 SDL_Flip(screen);
44         }
45         TTF_CloseFont(font);
46         TTF_Quit();
47         SDL_FreeSurface(text);
48         SDL_Quit();
49         return EXIT_SUCCESS;
50 }
```

الصورة التالية تمثّل النتيجة في غضون 13,9 ثانية بالتحديد :



لا تتردد في تنزيل المشروع إذا أردت دراسته بالتفصيل و تحسينه. هو ليس مثالياً بعد : يمكننا مثلاً استعمال `SDL_Delay` لتجنّب استعمال المعالج بنسبة 100%.

https://openclassrooms.com/uploads/fr/ftp/mateo21/ttf_exercice_temps.zip (437 Ko)

للذهاب بعيدا

إذا أردت التقدّم و تحسين هذا البرنامج، يمكنك أن تحاول صنع لعبة أين يجب النقر بالفأرة العدد الأقصى من المرات الممكنة في النافذة في وقت محدود حيث تتزايد قيمة العداد بعد كلّ نقرة.

يجب أن يتم إظهار عداد عكسي. حينما يصل إلى الصفر، نظهر عدد النقرات التي تم القيام بها و نطلب من المستعمل ما إن كان يريد إعادة المحاولة.

يمكنك أيضاً معالجة أفضل النتائج و تسجيلها في ملف. هذا سيساعدك في التدرب من جديد على استخدام الملفات في الـC.

حظاً موفقا !

ملخّص

- لا يمكننا أن نكتب نصاً في الـSDL، إلا إن استعملنا تمديداً كالمكتبة SDL_ttf.
- تسمح هذه المكتبة بتحميل ملفات خطوط ذات صيغة `.ttf` بالاستعانة بالدالة `TTF_OpenFont`.
- توجد ثلاث طرق لكتابة نص، ترتيبها من الأبسط إلى الأكثر تعقيدا : Solid، Shaded ثم Blended.
- يمكننا الكتابة في `SDL_Surface` عن طريق دوال مثل `TTF_RenderText_Blended`.

الفصل 27

تشغيل الصوت بـFMOD

منذ أن اكتشفنا الـSDL، تعلّمنا موضعة صور على النافذة، التفاعل مع المُستعمل بالفأرة و لوحة المفاتيح، كتابة نصوص، لكن ينقص أمر بالتأكيد : الصوت !

سيسدّ هذا الفصل ذلك النقص. بما أن الإمكانيّات التي توفّرها لنا الـSDL من ناحية الصوت محدودة جداً، سنكتشف هنا مكتبة متخصصة في الصوت : FMOD.

1.27 تثبيت FMOD

لماذا FMOD ؟

أنت تعرف ذلك الآن : الـSDL ليست فقط مكتبة رسومية. هي تسمح أيضاً بمعالجة الصوت عن طريق وحدة تسمّى SDL_audio. فلماذا إذاً سنحضّر مكتبة خارجية لا علاقة لها بالـSDL كـFMOD ؟

في الواقع هو اختيار قمتُ به بعد عدّة اختبارات. كان بإمكاني أن أشرح لك طريقة معالجة الصوت بالـSDL لكنّي فضّلت عدم فعل ذلك.
سأشرح موقفي أكثر.

لماذا قمتُ بتجنّب SDL_audio ؟

يعتبر التحكم في الصوت بالـSDL "منخفض المستوى". هذا يعني أنه يجب القيام بالعديد من التعامُلات الدقيقة كي نستطيع تشغيل الصوت. بمعنى آخر، سيكون الأمر صعباً و لا أجد ذلك ممتعاً. توجد مكتبات أخرى تسمح بتشغيل الصوت بشكل بسيط.

تذكير بسيط : مكتبة "منخفضة المستوى" هي مكتبة قريبة من الحاسوب. يجب أن نتعرّف إذا على قليل من العمل الداخلي للحاسوب كي نستفيد منها و يتطلب الأمر في الواقع وقتاً أكثر من الوقت اللازم للقيام بنفس الشيء مع مكتبة "عالية المستوى".
لا تنس أنّ كلّ شيء نسبيّ : لا توجد مكتبات منخفضة المستوى من جهة و أخرى عالية المستوى من جهة أخرى. هي فقط أكثر أو أقل من بعضها البعض في المستوى. مثلا، المكتبة FMOD عالية المستوى مقارنة بالوحدة SDL_audio من الـSDL.

تفصيل آخر مهم، تسمح الـSDL بتشغيل صوت بصيغة WAV فقط. صيغة الصوت هذه ليست مضغوطة. أي أن موسيقى من 3 دقائق تأخذ عشرات الميغا أوكتي، على عكس الصيغ المضغوطة مثل MP3 أو Ogg التي تحجز حجم ذاكرة أقلّ بكثير (من 2 إلى 3 ميغا أوكتي).

في الواقع، لو نفكّر في الأمر جيّداً، كان الأمر مشابهاً بالنسبة للصور، فالـSDL لا تتعامل إلا مع الصيغة BMP (صُور غير مضغوطة) بشكل مبدئي. مما استوجب علينا تسطيب مكتبة إضافية و هي SDL_image لنتمكّن من قراءة صيغ الصور الأخرى كـJPEG، PNG، GIF، إلخ.

اعلم أنه هناك مكتبة مكافئة بالنسبة للصوت و هي : SDL_mixer. هي قادرة على قراءة عدد كبير من صيغ الصوت، من بينها MP3، Ogg، Midi ... و رغم ذلك، لم أكلّمك عن هذه المكتبة. لماذا ؟

لماذا قمتُ بتجنّب SDL_mixer ؟

SDL_mixer هي مكتبة نضيفها للـSDL، بطريقة SDL_image. هي سهلة للاستعمال و تقرأ العديد من صيغ الصوت المختلفة. لكن، و بعد الاختبارات التي قمتُ بها، تبيّن لي أن هذه المكتبة تحتوي عِلَلا مزعجة بالإضافة إلى كونها محدودة من ناحية المزايا التي تمنحها.

من أجل هذه الأسباب توجّهت مباشرة إلى FMOD، مكتبة لا علاقة لها بالـSDL بالتأكيد، لكن لها الأفضلية كونها قوية و متداولا عليها.

تنزيل FMOD

إن كنت قد حكيت لك كل هذا، فهذا فقط لأخبرك بأن اختيار FMOD لم يكن عشوائياً. ببساطة هي أفضل مكتبة مجانية استطعت إيجادها.
كما أنها سهلة الاستخدام كـSDL_mixer بأفضلية لا يمكن تجاهلها : لا توجد بها عِلَل برمجية.

تسمح FMOD بالقيام بالعديد من الوظائف التي لا تسمح بها SDL_mixer، كالتأثيرات الصوتيّة ثلاثية الأبعاد.

> !
> FMOD هي مكتبة مجانية لكن ليست تحت رخصة LGPL على عكس الـSDL. هذا يعني أنه بإمكانك أن تستخدمها مادامت لم تحقق بها برامج مدفوعة. إذا أردت أن يكون البرنامج غير مجاني، يجب أن تدفع رسوماً لمؤلّف المكتبة (سأتركك تطّلع على الأسعار من خلال الموقع الرسمي لـFMOD).
> كثير من الألعاب التجارية تستعمل FMOD و من أشهر هذه الألعاب : Starcraft II، World of Warcraft : Cataclysm، Crysis 2، إلخ.

تتوفر العديد من نسخ FMOD، و النسخة الموجّهة إلى الاستعمال في أنظمة التشغيل المألوفة (GNU/Linux، Windows، Mac OS X، ...) تُدعى FMOD Ex Programmers API.

نزّل إذا نسخة FMOD Ex المناسبة لنظام التشغيل الخاص بك. خذ النسخة المسمّاة "مستقرة" (stable).

و تأكد بشكل خاص ما إن كان لديك نظام تشغيل 32 bits أو 64 bits (في Windows، قم بنقر يميني على جهاز الكمبيوتر (Computer) ثم في قسم الخصائص (Properties) تجد المعلومة اللازمة).

http://www.fmod.org/fmod-downloads.html#FMODExProgrammersAPI

تثبيت FMOD

يعمل التثبيت بنفس مبدأ عمل المكتبات السابقة، أي مثل الـSDL.

يجدر بالملف الذي حمّلته أن يكون ملفاً تنفيذياً (في Windows)، أو أن يكون أرشيفاً (`.dmg` في Mac OS X و `.tar.gz` في GNU/Linux).

1. ثبّت FMOD Ex على قرصك الصلب. الملفات التي نحتاجها يجب أن تتواجد في مجلّد يشبه هذا : `C:\Program Files\FMOD SoundSystem\FMOD Programmers API Win32\api`.
2. في هذا المجلّد تجد الـDLL الخاصّ بـFMOD Ex (`fmodex.dll`) و يجب أن يوضع في مجلّد المشروع. الـDLL الأخرى، أي `fmodexL.dll` تعمل على تنقيح العلل البرمجية. لن نقوم بذلك هنا. تذكّر فقط بأن الملف `fmodex.dll` هو الذي يجب أن تُعطيه مع الملف التنفيذي للبرنامج.
3. في المجلّد `api/inc`، تجد الملفات `.h`. ضعها كلّها إلى جانب الملفات الرأسية التي هي في مجلّد البيئة التطويرية. مثلا : `Code Blocks/mingw32/include/fmodex` (لقد أنشأت مجلّداً خصيصاً لأجل FMOD كما مثل الـSDL).
4. في المجلّد `api/lib`، استرجع الملف الموافق للمترجم. يجدر بملف نصّي أن يشير إلى أي ملف يجب أن نأخذ.

- إذا كنت تستعمل Code::Blocks، فالمترجم هو mingw. أنسخ الملف `libfmodex.a` في المجلّد `lib` للبيئة التطويرية.
  في Code::Blocks، إنه المجلّد `CodeBlocks/mingw32/lib`.
- إذا كنت تستعمل Visual C++، استرجع الملف `fmodex_vc.lib`.

5. أخيراً، الشيء الأكثر أهمية ربّما، يوجد مجلّد `documentation` في المجلّد FMOD Ex. من المفروض أن تتم إضافة اختصارات إلى قائمة "إبدأ" نحو هذه الملفات التوجيهية. أبق نظرك عليها لأنه لا يمكننا أن نكتشف كلّ ميزات FMOD Ex في هذا الفصل. ستحتاج إلى هذه الملفات في أقرب الآجال بالتأكيد.

يبقى أن نخصص المشروع. هنا أيضاً و مثل كلّ مرة : افتح المشروع بواسطة البيئة التطويرية المفضّلة و أضف الملف `.a` (أو `.lib`) إلى قائمة الملفات التي يجب أن يسترجعها محرر الروابط.
في Code::Blocks (يخالجني شعور بأنني أقوم بالتكرار)، إذهب إلى قائمة `Project` / `Build Options` ثم قسم `Linker`، أنقر على `Add` و أشر إلى المسار الذي يوجد به الملف `.a`. إذا ظهرت لك الرسالة : "Keep as a relative path ?"، أنصحك بأن تجيب بالسلب لكن يجدر بالأمور أن تشتغل في كلتا الحالتين.

تم تثبيت FMOD Ex، فلنرَى بسرعة مما هي مُشَكَّلة.

2.27 تهيئة و تحرير غرض نظامي

المكتبة FMOD Ex متوفّرة من أجل اللغتين C و ++C.
الشيء الخاص فيها هو أن مطوّري هذه المكتبة احتفظوا ببعض التناسق في "تركيب الكلمات" (Syntax) بين اللغتين.
الميزة الأولى هي أنه إذا تعلّمت التعامل مع FMOD Ex في لغة الـC ستتمكن من فعل ذلك في الـ++C بنسبة 95%.

تضمين الملف الرأسي

قبل كلّ شيء، يلزمك أن تقوم بتضمين الملف الرأسي الخاص بـFMOD. لا بأس في التذكير بكتابته :

```
1 #include <fmodex/fmod.h>
```

لقد وضعت هذا الملف في المجلّد الداخلي `fmodex`. عدّل على هذا السطر من الشفرة على حسب المسار الذي يتواجد به الملف عندك.
إذا كنت تعمل على GNU/Linux، يجدر بالتسطيب أن يتم تلقائيّا في المجلّد `fmodex`.

إنشاء و تهيئة غرض نظامي

الغرض النظامي هو عبارة عن متغير نستفيد منه على طول البرنامج لكي نعرّف معاملات المكتبة.
تذكّر أنه بالـSDL مثلاً، كان يجب أن نهيّئ المكتبة بشكل مباشر بواسطة دالة. هنا، دليل الاستعمال مختلف قليلاً : في عوض تهيئة كلّ المكتبة، لن نعمل إلا بغرض (Object) دوره تعريف سلوك هذه الأخيرة.

لكي ننشئ غرضا نظاميا، يكفي أن نعرّف مؤشّرا من نوع `FMOD_SYSTEM`. مثلا :

```
1 FMOD_SYSTEM *system;
```

لكي نحجز مكاناً في الذاكرة من أجل هذا الغرض النظامي، نستعمل الدالة `FMOD_System_Create` و التي نموذجها هو الآتي :

```
FMOD_RESULT FMOD_System_Create(FMOD_SYSTEM ▫▫ system);
```

لاحظ أن هذه الدالة تأخذ مؤشّرا نحو مؤشّر يؤشّر نحو `FMOD_SYSTEM` . القرّاء الأكثر حرصاً كانوا قد لاحظوا أنه لدى تعريف المؤشّر `FMOD_SYSTEM` ، لم يتم حجزه بواسطة `malloc` أو أي دالة أخرى. لهذا السبب تماماً تأخذ الدالة `FMOD_SYSTEM` معاملا من ذلك النوع لكي تحجز مكاناً للمؤشّر النظامي.

بعد تعريف الغرض النظامي ، تكفي كتابة :

```
FMOD_SYSTEM ▫system;
FMOD_System_Create(&system);
```

هكذا إذا، بما أننا نتوفّر الآن على الغرض النظامي، لم يتبقّ علينا سوى تهيئته. لفعل هذا، نستعمل الدالة `FMOD_System_Init` ذات النموذج :

```
FMOD_RESULT FMOD_System_Init(
        FMOD_SYSTEM ▫  system,
        int  maxchannels,
        FMOD_INITFLAGS  flags,
        void ▫  extradriverdata
);
```

- المعامل `system` هو المعامل الذي يهمّنا أكثر، لأنه المؤشّر الذي سنقوم بتهيئته.
- المعامل `maxchannels` يمثّل العدد الأقصى للقنوات التي يجب أن تديرها `FMOD` . بمعنى آخر، هو العدد الأقصى للأصوات التي يمكن أن يتم تشغيلها في نفس الوقت. هذا يعتمد على قوة بطاقة الصوت لديك . ننصح عادة بقيمة 32 (قيمة كافية من أجل معظم الألعاب البسيطة). لمعلوماتك، يمكن نظرياً لـ `FMOD` إدارة 1024 قناة مختلفة، لكن بهذا المستوى ستخاطر بجعل حاسوبك يشتغل كثيرا !
- المعامل `flag` لا يهمّنا كثيراً في هذا الفصل، سنكتفي بإعطائه القيمة `FMOD_INIT_NORMAL` .
- المعامل `extradriverdata` لا يهمّنا أيضاً، سنعطيه القيمة `NULL` .

مثلا، لكي نعرّف، نحجز، و نهيّئ غرضا نظاميا، نقوم بكتابة التالي :

```
FMOD_SYSTEM ▫system;
FMOD_System_Create(&system);
FMOD_System_Init(system, 2, FMOD_INIT_NORMAL, NULL);
```

نتوفّر الآن على غرض نظامي جاهز للإستعمال.

غلق و تحرير غرض نظامي

نغلق ثمّ نحرر الغرض النظامي بواسطة دالتين :

```
FMOD_System_Close(system);
FMOD_System_Release(system);
```

هل يجدر بي أن أعلّق على هذه الشفرة ؟

3.27 الأصوات القصيرة

فلنبدأ بدراسة الأصوات قصيرة المدّة.
"الصوت القصير" كما أسمّيه، هو صوت يستمرّ غالباً بضعة ثوانٍ (أحياناً أقل من ثانية) و غالباً ما يُوجه للاستعمال المنتظم.

أمثلة عن أصوات قصيرة :

- صوت إطلاق رصاصة.
- صوت مشي اللاعب.
- صوت tic-tac (لكيّ نوتّر اللاعب قبل انتهاء العد العكسي).
- صوت التصفيق.
- إلخ.

باختصار، كل صوت لا يعتبر موسيقى.
بشكل عام، هذه الأصوات قصيرة المدة إلى درجة أنه لا نحتاج إلى ضغطها. نجدها إذا في غالب الأحيان بصيغة WAV غير مضغوطة.

إيجاد الأصوات القصيرة

قبل أن نبدأ، سيكون من الجيد أن نتعرّف على بعض المواقع التي تقترح بنوكاً من الأصوات. بالفعل، لا أحد يريد أن يبدأ في تسجيل الأصوات بنفسه في المنزل.

سيكون الأمر جيداً فالأنترنت تقترح أصواتا قصيرة، غالباً بصيغة WAV.
أين نجدها ؟ قد يبدو الأمر سخيفاً، لا يجب علينا أن نفكّر في ذلك (مع أنه لازم)، لكن Google صديقنا. بشكل عشوائي، أكتب : "Free Sounds" و التي تعني "أصوات مجانية" بالإنجليزيّة، ستظهر لي ملايين النتائج.

لا شيء تحتاجه أكثر من الصفحة الأولى للبحث، ستجد ضالّتك هناك.
شخصياً حفظت الموقع FindSounds.com، محرّك بحث متخصص في الأصوات. لا أدري إن كان الأفضل، لكن على أي حال هو موقع كامل.

م

إذا لم تعرف ماهي الكلمات المفتاحية التي تستعملها في البحث، توجّه إلى الصفحة الخاصة بأمثلة عن الكلمات المفتاحية للبحث. يجب عليك أن تجيد بعض الكلمات الإنجليزية بالتأكيد (لكن على أي حال إن كنت تريد أن تصبح مبرمجاً، كيف ستفعل لو أنك لا تجيد على الأقل اللغة الإنجليزية ؟).

بالبحث عن كلمة "gun"، سنجد أطنانا من أصوات إطلاق النار بالبندقية، لو نكتب "door" سنجد أصوات تحرّك الباب (الصورة التالية)، إلخ.



الخطوات التي يجب إتّباعها لتشغيل الصوت

الخطوة الأولى تنصّ على تحميل الصوت الذي نريد تشغيله في الذاكرة.
أنصحك بتحميل كلّ الأصوات التي ترى أنك ستستعملها كثيراً منذ بداية البرنامج. تقوم بتحريرها في النهاية. في الواقع، ما إن يتم تحميل الصوت في الذاكرة، ستكون قراءته سريعة جداً.

المؤشّر

الخطوة الأولى : إنشاء المؤشّر من نوع `FMOD_SOUND` و الذي يمثّل الصوت.

```
FMOD_SOUND *fire = NULL;
```

تحميل الصوت

الخطوة الثانية : تحميل الصوت بواسطة الدالة `FMOD_System_CreateSound` . هي تأخذ ... خمسة معاملات :

- **الغرض النظامي** الذي تحدّثنا عنه سابقاً.
  بالطبع يجب أن يكون هذا الغرض جاهزاً للاستعمال (معرّفا، محجوزاً و مهيّئاً).
- **اسم الملف الصوتي** الذي نريد تحميله. يمكن أن يكون ذو صيغة WAV، MP3، OGG، إلخ. من المستحسن دائماً أن يتم تحميل أصوات قصيرة (بضع ثوانٍ كحدّ أقصى) على أن يتم تحميل أصوات طويلة. في الواقع، تحمّل الدالة و تفك تشفير كلّ الصوت في الذاكرة، مما قد يأخذ مكانا كبيرا لو أن الصوت هو موسيقى !
- المعامل الثالث هو علَم.

يهمّنا بشكل خاص هنا لأنه بفضله يمكننا أن نقول لـFMOD أن الصوت الذي ستشغّله هو عبارة عن صوت قصير. من أجل هذا نستعمل القيمة `FMOD_CREATESAMPLE` .

- و الرابع لا يهمّنا، سنعطيه القيمة `NULL` .
- المعامل الأخير هو من نوع `FMOD_SOUND ** sound` ، و هذا المؤشّر سنستعمله لاحقاً من أجل تشغيل الصوت.
  بشكل ما، يمكننا القول بأن هذا المؤشّر سيؤشّر على الصوت الذي نريد تشغيله.

هذا مثال عن تحميل :

```
1 FMOD_System_CreateSound(system, "pan.wav", FMOD_CREATESAMPLE, 0, &fire);
```

هنا، أقوم بتحميل الصوت `pan.wav` . المؤشّر `fire` سيعتبر كمرجع للصوت لاحقاً.

> م
> إذا كنت تريد أن تختبر الشفرات في نفس الوقت الذي أعطيها لك، أنصحك بتنزيل الصوت `pan.wav` (http://www.siteduzero.com/uploads/fr/ftp/mateo21/pan.wav)
> و الذي سأستعمله أيضاً في بقيّة هذا الفصل.

إذا اشتغل كلّ شيء على ما يُرام، تُرجع الدالة القيمة `FMOD_OK` و إلا، فهذا يعني أنه حدث مشكل خلال فتح الملف الصوتي (ملف تالف أو غير موجود مثلا).

تشغيل الصوت

تريد تشغيل الصوت ؟ لا يوجد مشكل مع الدالة `FMOD_System_PlaySound` !

يكفي أن تعطيها غرضا نظاميا جاهزا للاستعمال، رقم القناة التي نريد أن يُلعب فيها الصوت و أيضاً المؤشّر نحو الصوت، إضافة إلى معاملات أخرى لا تهمّنا سنعطيها القيمة `NULL` أو 0. بالنسبة لرقم القناة، لا تشغل بالك بالتفكير و ابعث القيمة `FMOD_CHANNEL_FREE` و اترك FMOD تتحكّم في ذلك.

```
1 FMOD_System_PlaySound(system, FMOD_CHANNEL_FREE, fire, 0, NULL);
```

تحرير الصوت من الذاكرة

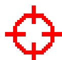
حينما تصبح غير محتاجٍ للصوت، يجب عليك تحريره.
لا يوجد ما هو أسهل، يكفي أن تشير إلى المؤشّر الذي تريد تحريره بواسطة الدالة `FMOD_Sound_Release`.

مثال : لعبة إطلاق النار

الأفضل الآن هو أن نلخّص كل ما تعلّمناه، عن طريق مثال واضح عن برنامج مكتوب بالـSDL.
لا يوجد شيء معقّد هنا و يجدر ألا تصادفك أية مشكلة في تحقيق هذا التمرين.

الموضوع

مهمتك سهلة : إنشاء لعبة إطلاق النار.
حسناً، لن ننشئ لعبة كاملة هنا، لكننا سنتحكّم في المصوّب. لقد صممتُ لك مصوّباً بسيطاً بواسطة برنامج الرسام :



باختصار، إليك المهام :

- خلفية النافذة : سوداء.
- مؤشّر الفأرة : غير مرئي.
- يتم تسوية صورة المصوّب على وضعية الفأرة حينما نقوم بتحريكه. احذر : يجب أن يتم لصق مركز الصورة على مستوى مؤشّر الفأرة.
- حينما ننقر بالفأرة، يجب تشغيل الصوت `pan.wav`.

قد تكون هذه البداية لصنع لعبة إطلاق نار كاملة.
سهل جداً ؟ حسناً، حان وقت العمل إذا !

التصحيح

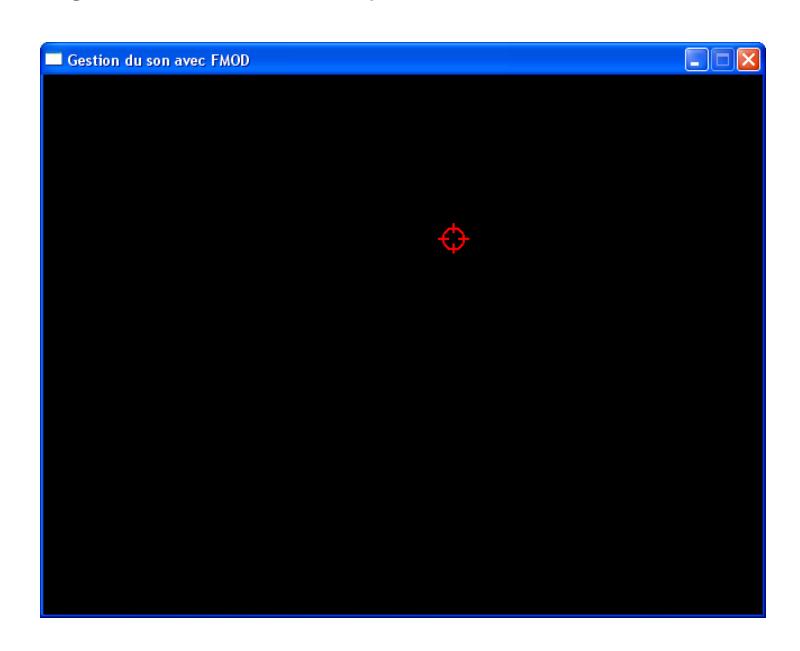
هذه هي الشفرة المصدرية الكاملة :

```c
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
#include <SDL/SDL_image.h>
#include <fmodex/fmod.h>
int main(int argc, char *argv[])
```

```c
{
        SDL_Surface *screen = NULL, *gunSight = NULL;
        SDL_Event event;
        SDL_Rect position;
        int cont = 1;
        FMOD_SYSTEM *system;
        FMOD_SOUND *fire;
        FMOD_RESULT result;
        // Initializing and creating a system object
        FMOD_System_Create(&system);
        FMOD_System_Init(system, 1, FMOD_INIT_NORMAL, NULL);
        // Loading the sound and checking the loading
        result = FMOD_System_CreateSound(system, "pan.wav", FMOD_CREATESAMPLE, 0, &fire);
        if (result != FMOD_OK)
        {
                fprintf(stderr, "Can't read pan.wav\n");
                exit(EXIT_FAILURE);
        }
        // Initializing the SDL
        SDL_Init(SDL_INIT_VIDEO);
        SDL_ShowCursor(SDL_DISABLE);
        screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
        SDL_WM_SetCaption("Gestion du son avec FMOD", NULL);
        gunSight = IMG_Load("viseur.png");
        while (cont)
        {
                SDL_WaitEvent(&event);
                switch(event.type)
                {
                        case SDL_QUIT:
                        cont = 0;
                        break;
                        case SDL_MOUSEBUTTONDOWN:
                        // When we click, we play the sound
                        FMOD_System_PlaySound(system, FMOD_CHANNEL_FREE , fire, 0, NULL);
                        break;
                        case SDL_MOUSEMOTION:
                        // When we move the mouse, we move the gun sight too. To do this we have to use gunSight->w/2, gunSight->h/2
                        position.x = event.motion.x - (gunSight->w / 2);
                        position.y = event.motion.y - (gunSight->h / 2);
                        break;
                }
                SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 0, 0, 0));
                SDL_BlitSurface(gunSight, NULL, screen, &position);
                SDL_Flip(screen);
```

```
52          }
53          // We close the SDL
54          SDL_FreeSurface(gunSight);
55          SDL_Quit();
56          // We free the sound, free and close the system object
57          FMOD_Sound_Release(fire);
58          FMOD_System_Close(system);
59          FMOD_System_Release(system);
60          return EXIT_SUCCESS;
61  }
```

الصورة التالية تعطيك لمحة عن اللعبة المصغّرة، لكن الأفضل أن ترى الفيديو بالصوت على الأنترنت :



مشاهدة الفيديو هنا :
https://openclassrooms.com/uploads/fr/ftp/mateo21/viseur.html

هنا، حمّلت FMOD قبل الـSDL و حررتها بعدها. لا توجد قواعد من ناحية الترتيب (كان بإمكاني القيام بالعكس). اخترتُ تحميل الـSDL و فتح النافذة بعد تحميل FMOD لكي تكون اللعبة جاهزة للاستعمال ما إن يتم فتح النافذة (و إلا كان بالإمكان أن تنتظر بعض الميلي ثواني ريثما يتم تحميل FMOD).
و مع ذلك أنت حر باختيار الترتيب، هذه تفاصيل ليس إلا.

أعتقد أنني علّقت كفاية على الشفرة. لا يوجد فخ معيّن، و لا يوجد جديد يذكر. يمكننا أن نذكر الصعوبة "الصغيرة" التي تنص على لصق مركز المصوّب على مستوى مؤشّر الفأرة. يتم حساب وضعية الصورة بدلالة ذلك.

بالنسبة لمن لم يفهم الفرق بعد، سنرى ذلك الآن. بالمناسبة، لقد أعدت تفعيل الإظهار الخاص بالمؤشّر كي نرى كيف يتموضع المصوّب بالنسبة لمؤشّر الفأرة.

شفرة خاطئة (المصوب في وضعية خاطئة)		`position.x = event.motion.x;` `position.y = event.motion.y;`
شفرة صحيحة (المصوب في وضعية صحيحة)		`position.x = event.motion.x - (gunSight->w / 2);` `position.y = event.motion.y - (gunSight->h / 2);`

أفكار للتحسين

هذه اللعبة تعتبر قاعدة للعبة إطلاق النار. لديك المصوّب، صوت إطلاق النار، لم يتبقّ لك سوى إظهار أو تمرير أعداء لكي يقوم اللاعب بإطلاق النار عليهم ثم يتم تسجيل النتيجة. كالعادة، عليك بالعمل وحدك. تريد صنع لعبة ؟ لا شيء يمنعك، بل لديك المستوى الكافي الآن، و لديك أيضاً شفرة مبدئية للعبة إطلاق النار ! ماذا تنتظر، صدقاً ؟

> طبعاً، منتديات OpenClassrooms ستساعدك في حال علقت في أيّ لحظة أثناء إنشائك للعبة. مواجهة الصعوبات أمر عادي مهما كان المستوى الذي أنت فيه.

4.27 الموسيقى (WAV، MP3، OGG)

نظرياً، يسمح العَلم `FMOD_CREATESAMPLE` بتحميل أي صوت مهما كان، من ضمنها الصيغ المضغوطة WAV، MP3، OGG. المشكل يخصّ الأصوات "الطويلة"، أي الموسيقى.

في الواقع، تدوم الموسيقى كمعدّل من 3 إلى 4 دقائق. لكن، بهذا العلم، تحمّل الدالة `FMOD_System_CreateSound` كلّ الملف في الذاكرة (و النسخة غير المضغوطة هي التي ستتواجد في الذاكرة، فهذا يعني أنها ستأخذ حيّزاً كبيراً !).

إذا كان لديك صوت طويل المدّة (سنتكلّم عن "الموسيقى" من الآن و صاعداً)، سيكون من المستحسن أن نحمّلها بشكل تدفّقي (streaming)، يعني أن نحمّل منها أجزاء صغيرة في نفس الوقت الذي تشغّل فيه، هذا في الواقع ما تقوم به كلّ البرامج الخاصة بقراءة الأصوات.

إيجاد الموسيقى

ندخل هنا إلى أرضية ملغّمة، شائكة، بها متفجّرات (سمّها كما تريد).
في الواقع، أغلب الموسيقى و الأغاني التي نعرفها معنيّة بحقوق المؤلّف. حتى لو كتبت برنامجاً صغيراً، يجب أن تدفع رسوماً إلى الـSACEM (في فرنسا على الأقل).

إذاً، على غرار الموسيقى MP3 المحميّة بحقوق المؤلّف، ماذا يتبقّى لنا ؟
لحسن الحظ، توجد أغاني حرة من الحقوق ! يسمح أصحابها بنشر أغانيهم بشكل حرّ، إذا لا يوجد أي مشكل في استعمالها في برامجك.

!

إذا كان برنامجك تجاريا، يجب أن تتكلّم مع الفنان نفسه، فهناك من لا يقبل الاستعمالات التجارية لأغانيه. الأغنية حرّة الحقوق يمكن تنزيلها، نسخها و سماعها بشكل حرّ، لكن هذا لا يعني أنه بإمكاننا تحصيل المال على حساب الفنانين !

حسناً السؤال هو : "أين نجد موسيقى حرّة ؟". يمكننا أن نجري بحثاً بالعبارة "Free Music" في Google، لكن هنا، هو ليس صديقنا هذه المرة. في الواقع، لنعرف لماذا، لقد كتبنا الكلمة Free لكننا سنقع دائما على مواقع تطلب منا اشتراء الأغاني !

لحسن الحظ، توجد مواقع تحتوي موسيقى حرة الحقوق. هنا أنصحك بالموقع الجيد Jamendo لكنه ليس وحده الموجود في المجال. https://www.jamendo.com/

يتم تقسيم الأغاني على حسب النمط. لديك الكثير من الخيارات، ستجد الجيد، السيء و الجيد جداً، و عديم الجودة. في الواقع، كلّ يعتمد على ذوقك و تقبّلك للأنماط المختلفة من الموسيقى. من المستحسن أن تختار موسيقى تشغّل في خلفية اللعبة و تكون متناسبة مع عالم اللعبة.

لمعلوماتك، توجد أغنية تنتمي إلى الألبوم "Lies and Speeches" للمجموعة "Hype". إذا أردت معرفة المزيد عنها، زر صفحتهم على الموقع My Space : http://www.myspace.com/hypemusic

م

متيقّن جداً أن الأذواق و الألوان لا مجال للنقاش فيهما. لا بأس باختيارك لأغنية أخرى إن كانت هذه لا تُعجبك.

لقد حمّلت إذاً الألبوم و سأستعمل أغنية Home بصيغة MP3.
يمكنك تنزيلها مباشرة إذا أردت إختبار الشفرة في نفس الوقت معي. هذه من ميزات الموسيقى الحرة : يمكننا نسخها، توزيعها بشكل حر و لهذا لن نكون منزعجين.

الخطوات الواجب اتّباعها لتشغيل الموسيقى

الاختلاف الوحيد هو العَلم المُعطى للدالة `FMOD_System_CreateSound`.
في عوض أن نعطيها العَلم `FMOD_CREATESAMPLE`، نعطيها الأعلام : `FMOD_SOFTWARE`، `FMOD_2D` و `FMOD_CREATESTREAM`.
لا تنتظر كثيراً أن أشرح لك بشكل موسّع معاني هذه الأعلام، العَلم الذي يهمّنا أكثر هو `FMOD_CREATESTREAM` لأنه يطلب من FMOD تحميل الموسيقى جزءاً بجزء.

لكي نستعمل كلّ هذه الأعلام في نفس الوقت، نستعمل العامل المنطقي `|` بهذه الطريقة :

```
1 FMOD_System_CreateSound(system, "my_music.mp3", FMOD_SOFTWARE | FMOD_2D |
     FMOD_CREATESTREAM, 0, &sound);
```

هكذا هو العمل !

لكن هذا ليس كلّ شيء. في حالة موسيقى، سيكون من الجيد أن نعرف كيف نغيّر من قوّة الصوت، نتحكّم في إعادة الأغنية مرات عديدة، إيقافها مؤقّتا، أو حتى إيقافها كلياً. هذا النوع من الأشياء ما سنراه الآن. لكن قبل هذا، سنحتاج أن نعمل على القنوات بشكل مباشر.

استرجاع قناة أو مجموعة من القنوات

في نسخ سابقة من المكتبة FMOD، رقم الهويّة البسيط لقناة يكفي لكي يغيّر قوّة الصوت أو إيقاف أغنية مؤقّتا.
طرأ تغيير صغير منذ FMOD Ex : من خلال رقم القناة، نستعمل دالة توفّر لنا مؤشّراً نحو القناة. الفكرة تبقى نفسها، تتغيّر طريقة التنفيذ فقط.
يتم تعريف قناة كنوع `FMOD_CHANNEL` و الدالة التي تسمح باسترجاع قناة إنطلاقاً من رقم الهوية هي `FMOD_System_GetChannel`.
مثلاً، إذا كان لدي غرض نظامي و أردت استرجاع القناة رقم 9، يجب أن أكتب :

```
FMOD_CHANNEL *channel;
FMOD_System_GetChannel(system, 9, &channel);
```

لا شيء أسهل من هذا !

- المعامل الأول هو الغرض النظامي.
- المعامل الثاني هو رقم الهويّة الخاص بالقناة.
- المعامل الثالث هو عنوان المؤشّر الذي نريد تخزين المعلومة المُرادة فيه.

ما إن نحصّل على مؤشّر القناة، يمكننا بسهولة التعامل مع الموسيقى (تغيير قوة الصوت، إيقاف الموسيقى مؤقّتا، ...).

لاحظ أنه يمكننا أيضاً استرجاع مجموعة كاملة من القنوات في مؤشّر واحد : بهذا نتجنّب أن نقوم بنفس العملية من أجل كلّ قناة مختلفة.
نوع مجموعة القنوات هو `FMOD_CHANNELGROUP` وواحدة من الدوال التي تهمّنا أكثر هي `FMOD_System_GetMasterChannelGroup` لأنها تسمح بالحصول على مؤشّر نحو كلّ القنوات المستعملة من طرف الغرض النظامي.
أسلوب عمل هذه الدالة مماثل لسابقتها.

تغيير قوة الصوت

لنغيّر قوة الصوت، يمكننا أن نقوم بذلك إما من أجل قناة محددة أو من أجل كلّ القنوات.
مثلاً، لكي نقوم بذلك من أجل كلّ القنوات، يجب أوّلاً استرجاع المؤشّر نحو مجموعة القنوات، ثم استعمال الدالة `FMOD_ChannelGroup_SetVolume` ذات النموذج :

```
FMOD_ChannelGroup_SetVolume
FMOD_RESULT FMOD_ChannelGroup_SetVolume(
        FMOD_CHANNELGROUP *  channelgroup,  float  volume
);
```

المعامل `channelgroup` هو المعامل الذي نحن بصدد استرجاعه.
المعامل `volume` من نوع `float`، حيث 0.0 توافق المستوى الصامت و 1.0 توافق قراءة بكامل قوّة الصوت (هذه القيمة هي القيمة المُختارة تلقائيا).

إعادة تشغيل الأغنية

غالباً ما نحتاج إلى إعادة تشغيل الموسيقى الخلفية. هذا تماما ما تقترحه الدالة `FMOD_Sound_SetLoopCount` و التي تأخذ معاملين :

- المؤشّر نحو الأغنية.
- عدد المرات التي يجب فيها أن تتم إعادة قراءة الموسيقى. إذا وضعت القيمة 1، تم إذا قراءة الأغنية مرتين. إذا وضعت قيمة سالبة (مثل -1)، تتم إعادة قراءة الأغنية إلى ما لانهاية.

بهذه الشفرة المصدرية، يتم تكرار الأغنية إلى ما لانهاية :

```
FMOD_Sound_SetLoopCount(music, -1);
```

> X لكي تشتغل عملية إعادة التشغيل، يجب أن نبعث `FMOD_LOOP_NORMAL` كمعامل ثالث للدالة `FMOD_System_CreateSound`.

إيقاف الموسيقى مؤقتا

توجد هنا دالتان لكي تتعلّمهما :

- `FMOD_Channel_GetPaused` : تشير ما إذا كانت الأغنية التي يتم تشغيلها حالياً في القناة المختارة في حالة متوقفة مؤقتا أم لا. تعطي القيمة "صحيح" للمتغير `state` إذا كانت الأغنية متوقفة مؤقتا، و القيمة "خطأ" ما إن كانت تشتغل حالياً.
- `FMOD_Channel_SetPaused` : توقف الأغنية مؤقتا أو تعيد تشغيلها في القناة المشار إليها. أعطها القيمة 1 (صحيح) لإيقافها مؤقتا. و 0 (خطأ) لإعادة تفعيل القراءة.

هذه الشفرة المصدرية الخاصة بنافذة SDL تقوم بإيقاف الأغنية مؤقتا إذا ضغطنا على الزر `P` من لوحة المفاتيح، و تعيد تفعيلها إذا ضغطنا مجدداً على `P`.

```
case SDL_KEYDOWN:
if (event.key.keysym.sym == SDLK_p) // If we press P
{
        FMOD_BOOL state;
        FMOD_Channel_GetPaused(channel, &state);
        if (state == 1) // if the music is paused
                FMOD_Channel_SetPaused(channel, 0); // We play it
        else // Else, the music is being played
                FMOD_Channel_SetPaused(channel, 1); // We pause it
}
break;
```

إذا أردنا إعادة تطبيق نفس الأمر من أجل كلّ القنوات معاً، نستعمل الدالتين `FMOD_ChannelGroup_GetPaused` و `FMOD_ChannelGroup_SetPaused`، الاختلاف الوحيد الذي يجب القيام به هو إعطاءها كمعامل `FMOD_CHANNELGROUP` بدلاً عن `FMOD_CHANNEL`.

إيقاف التشغيل

يكفي استدعاء `FMOD_Channel_Stop` لأجل إيقاف موسيقى في قناة ما، أو `FMOD_ChannelGroup_Stop` من أجل مجموعة من القنوات، نبعث لها على الترتيب المؤشر نحو القناة أو المؤشّر نحو مجموعة القنوات.

و بالطبع أشياء أخرى

يمكننا القيام بالكثير من الأشياء الأخرى، لكن لن أقوم بتعدادها كلّها هنا. يجب عليك قراءة الملفات التوجيهية ! و التي أنصحك بإلقاء نظرة عليها في حال ما احتجت الاطلاع على دوال أخرى.

تحرير الذاكرة

لكي نقوم بتفريغ الأغنية من الذاكرة، نستدعي الدالة `FMOD_Sound_Release` و نعطيها المؤشّر :

```
FMOD_Sound_Release(music);
```

الشفرة المصدرية الكاملة لقراءة ملف MP3

الشفرة أسفله تقدّم لنا برنامجاً يقوم بتشغيل الموسيقى "Home" التي حصلنا عليها من الموقع Jamendo.
يتم تشغيل الموسيقى منذ بداية تشغيل البرنامج. يمكننا إيقاف الموسيقى مؤقتا بالضغط على P.

```c
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
#include <SDL/SDL_image.h>
#include <fmodex/fmod.h>
int main(int argc, char *argv[])
{
        SDL_Surface *screen = NULL, *wallet = NULL;
        SDL_Event event;
        SDL_Rect position;
        int cont = 1;
        FMOD_SYSTEM *system;
        FMOD_SOUND *music;
        FMOD_RESULT result;
        FMOD_System_Create(&system);
        FMOD_System_Init(system, 1, FMOD_INIT_NORMAL, NULL);
        // We open the music
        result = FMOD_System_CreateSound(system, "hype_home.mp3", FMOD_SOFTWARE
             | FMOD_2D  | FMOD_CREATESTREAM, 0, &music);
        // We check if it has been opened successfully (IMPORTANT)
        if (result != FMOD_OK)
        {
                fprintf(stderr, "Can't read the mp3 file\n");
                exit(EXIT_FAILURE);
        }
        // We activate the music repetition infinitely
        FMOD_Sound_SetLoopCount(music, -1);
        // We play the music
        FMOD_System_PlaySound(system, FMOD_CHANNEL_FREE, music, 0, NULL);
        SDL_Init(SDL_INIT_VIDEO);
        screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
        SDL_WM_SetCaption("Gestion du son avec FMOD", NULL);
        wallet = IMG_Load("hype_liesandspeeches.jpg");
        position.x = 0;
        position.y = 0;
        while (cont)
        {
                SDL_WaitEvent(&event);
                switch(event.type)
                {
                        case SDL_QUIT:
                        cont = 0;
                        break;
                        case SDL_KEYDOWN:
                        if (event.key.keysym.sym == SDLK_p) // If we press P
```

```
45                         {
46                                 FMOD_CHANNELGROUP *channel;
47                                 FMOD_BOOL state;
48                                 FMOD_System_GetMasterChannelGroup(system, &
                                       channel);
49                                 FMOD_ChannelGroup_GetPaused(channel, &state);
50                                 if (state) // If the music is paused
51                                         FMOD_ChannelGroup_SetPaused(channel, 0)
                                               ; // We play it
52                                 else // Else, the music is being played
53                                         FMOD_ChannelGroup_SetPaused(channel, 1)
                                               ; // We pause it
54                         }
55                         break;
56                 }
57                 SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 0, 0, 0))
                       ;
58                 SDL_BlitSurface(wallet, NULL, screen, &position);
59                 SDL_Flip(screen);
60         }
61         FMOD_Sound_Release(music);
62         FMOD_System_Close(system);
63         FMOD_System_Release(system);
64         SDL_FreeSurface(wallet);
65         SDL_Quit();
66         return EXIT_SUCCESS;
67 }
```

لكي لا تكون خلفية البرنامج مجرّد صورة سوداء استعملت صورة الألبوم كخلفيّة.

يمكنك مشاهدة الفيديو الذي يمثّل تشغيل البرنامج من هنا.

https://openclassrooms.com/uploads/fr/ftp/mateo21/musique_hype.html (730 Ko)

ملخّص

- لدى الـSDL مزايا محدودة بالنسبة للصوت و يُنصح أن تتم الاستعانة بمكتبة مخصصة لتشغيل الصوت مثل FMOD.
- نميّز بين نوعين من الصوت بالFMOD : أصوات قصيرة (ضجيج الخطوات مثلا) و أصوات طويلة (موسيقى مثلاً).
- كلّ من هذين النوعين يُقرأ بنفس الدالة لكن بواسطة أعلام مختلفة كخيارات.
- تسمح FMOD بتشغيل كثير من الأصوات في آن واحد بالإستعانة بالكثير من القنوات.

الفصل 28

عمل تطبيقي : الإظهار الطيفي للصوت

هذا العمل التطبيقي سيقترح عليك التعامل مع الـSDL و الـFMOD في نفس الوقت. هذه المرّة، لن نعمل على لعبة. كما نعرف فالـSDL مخصصة لهذا، لكن يمكن استعمالها في ميادين أخرى. سيقوم هذا الفصل بإثبات أنها صالحة لأجل أشياء أخرى.

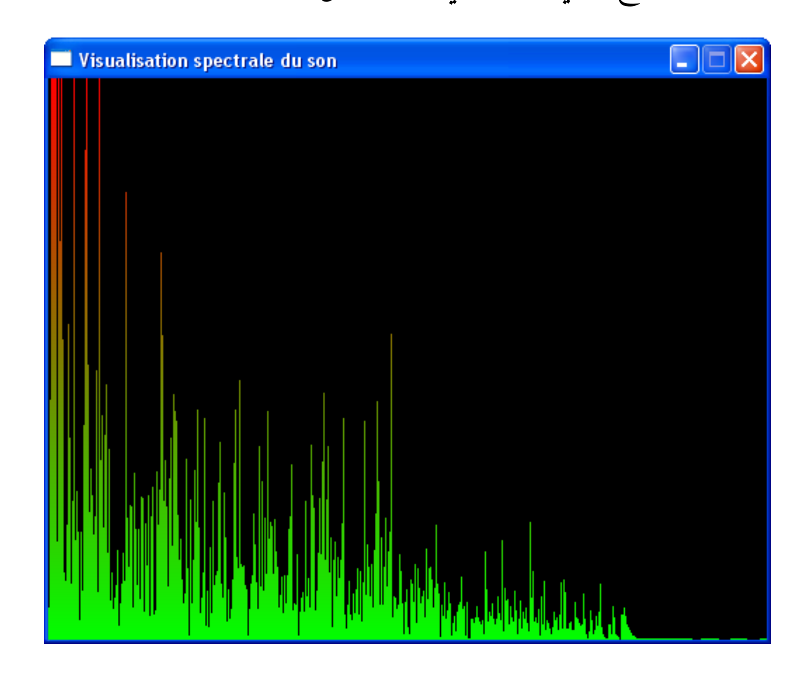
سنحقق هنا إظهاراً للطيف الصوتي بالـSDL. يتوقّف هذا على إظهار تركيبة الصوت الذي نشغّله، مثلاً موسيقى. نجد هذه الخاصية في كثير من برامج قراءة الأصوات. إنه أمرٌ ممتع و ليس بقدر الصعوبة التي يبدو عليها !

سيسمح لك هذا الفصل بالعمل على مفاهيم قُمنا باستكشافها مؤخّراً :

- التحكّم في الوقت.
- المكتبة FMOD.

سنتعرّف علاوة على ذلك، على كيفية التعديل على مساحة بيكسلا ببيكسل.

الصورة التالية تعطيك مظهراً للبرنامج الذي سنكتبه في هذا الفصل.



هو نوع الإظهار الذي نجده في قارئي الأصوات كـWinamp، Windows Media Player أو AmaroK.
كما قلتُ لك إن الأمر ليس صعبَ التحقيق. على عكس العمل التطبيقي الخاص بـMario Sokoban، هذه المرّة ستقوم بنفسك بالعمل. سيمثّل هذا بالنسبة إليك تمريناً جيداً.

1.28 التعليمات

التعليمات بسيطة. إتبّعها بخطوة بالترتيب، و لن تواجه أي مشاكل.

قراءة ملف MP3

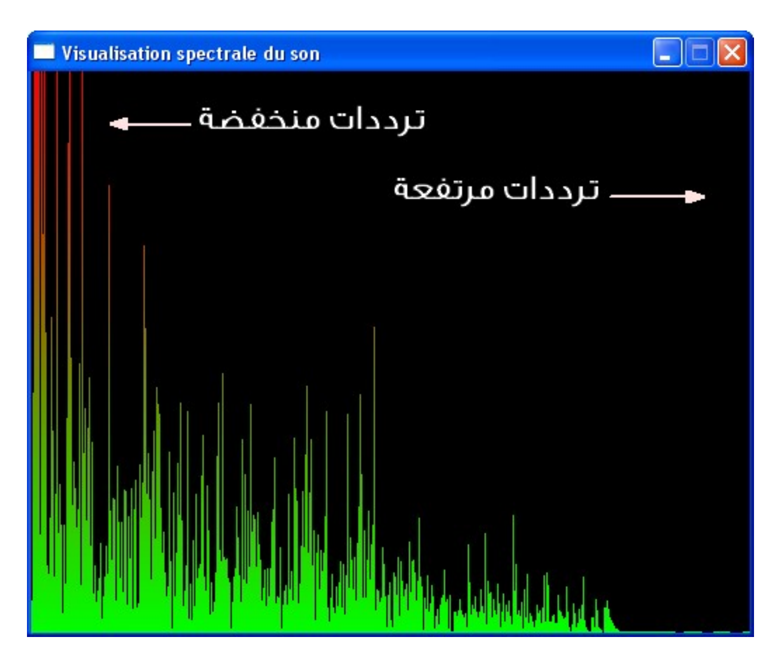
لكي تبدأ، يجب عليك إنشاء برنامج يقوم بقراءة ملف MP3. ليس عليك سوى إعادة الأغنية "Home" للمجموعة "Hype" و التي استعملناها في الفصل الخاص بـFMOD لتلخيص كيفية عمل تشغيل الموسيقى.

إذا اتّبعت جيّدا الفصل حول FMOD، لا تحتاج أكثر من بضعة دقائق لكي تقوم بالعملية. أنصحك بالمناسبة أن تقوم بنقل الملف MP3 إلى مجلّد المشروع.

استرجاع المعلومات الطيفية للصوت

لكي نعرف كيف يعمل الإظهار الطيفي للصوت، من الواجب أن أشرح لك كيفية يعمل الأمر من الداخل (بشكل تقريبي فقط، و إلا سندخل في درس رياضيات).

يمكن أن يتم تقسيم الصوت إلى ترددات (Frequencies). بعض الترددات منخفضة، بعضها متوسطة و بعضها مرتفعة.
ما سنقوم به في عملية الإظهار هو إظهار كمية كلّ واحدة من الترددات على شكل شرائط و كلّما يكون الشريط كبيراً، كلما يكون التردد مستعملاً أكثر :



على يسار النافذة، نقوم بإظهار الترددات المنخفضة، و على اليمين الترددات المرتفعة.

> ؟ لكن كيف نسترجع كمّيّة كلّ تردد ؟

ستهتم FMOD بهذا العمل. يمكننا استدعاء الدالة `FMOD_Channel_GetSpectrum` ذات النموذج :

```
FMOD_RESULT FMOD_Channel_GetSpectrum(
        FMOD_CHANNEL ▫  channel,
        float ▫  spectrumarray,
        int   numvalues,
        int   channeloffset,
        FMOD_DSP_FFT_WINDOW  windowtype
);
```

و هاهي المعاملات التي تحتاجها الدالة :

- القناة التي تشتغل فيها الموسيقى. يجب إذا استرجاع مؤشّر نحو هذه القناة.
- جدول `float`. يجب أن يتم حجز الذاكرة من أجل هذا الجدول مسبّقاً، بشكل ثابت أو حيّ، لكي نسمح لFMOD بملئه بشكل صحيح.
- حجم الجدول. يجب أن يكون حجم الجدول إجبارياً عبارة عن قوّة للعدد 2، مثلا 512.
- يسمح هذا المعامل بتعريف بأي مخرج نحن مهتمون. مثلاً لو أننا في stereo، فـ0 تعني اليسار و 1 تعني اليمين.
- هذا المعامل معقّد قليلاً، و لا يهمّنا حقيقة في هذا الفصل. سنكتفي بإعطائه القيمة `FMOD_DSP_FFT_WINDOW_RECT`.

> م تذكير : النوع `float` هو نوع عشري، مثل `double`. الاختلاف بين الاثنين يكمن في كون الـ`double` أكثر دقّة من الآخر، لكن في حالتنا يكفينا الـ`float`. هذا الأخير مستعمل من طرف FMOD هنا. و لذلك، هو ما سنستعمله نحن أيضاً.

بشكل واضح، نعرّف جدول الـ`float` :

```
float spectrum[512];
```

ثم، حين يتم تشغيل الموسيقى، نطلب من FMOD ملء جدول الأطياف بالقيام مثلاً بـ :

```
FMOD_Channel_GetSpectrum(channel, spectrum, 512, 0, FMOD_DSP_FFT_WINDOW_RECT);
```

يمكننا بعد ذلك تصفّح الجدول لكي نحصّل على قيم الأطياف :

```
spectrum[0] // The lowest frequency (Left)
spectrum[1]
spectrum[2]
...
spectrum[509]
spectrum[510]
spectrum[511] // The highest frequency (Right)
```

كلّ تردد هو عبارة عن عدد عشري محصور بين 0 (لا شيء) و 1 (قيمة قصوى). ينصّ عملك على إظهار كلّ شريط سواء كان قصيراً أو كبيراً بدلالة القيمة التي تحتويها كلّ من خانات الجدول.

مثلاً، إذا كانت القيمة هي 0.5 يجدر بك رسم شريط يكون علّوه مساوياً لنصف علوّ النافذة. إذا كانت القيمة هي 1، فسيأخذ الشريط كلّ علو النافذة.

بشكل عام، تكون القيم ضعيفة (أكثر قرباً من 0 على 1). أنصحك بضرب كلّ القيم بـ20 لكي ترى الطيف بشكل أفضل.
احذر : إذا قمت بهذا، تأكد بأنك لن تتجاوز 1 (قم بتدوير القيمة إلى 1 إذا احتجت إلى ذلك). إذا وجدت أنك تتعامل مع أعداد تفوق 1، فقد تواجه مشاكل لاحقاً في رسم الشرائط العموديّة لاحقا !

؟ لكن يجدر بالشرائط أن تتحرّك في نفس الوقت الذي يتم فيه تشغيل الصوت، أليس كذلك ؟ بما أن الصوت يتحرّك كلّ الوقت، يجب تحديث الصورة الرسومية، ما العمل ؟

سؤال جيد. في الواقع، الجدول الخاص المتكون من 512 `float` الذي ترجعه لنا FMOD يتغيّر كل 25 مث (لكي نكون في نفس الفاصل الزمني بالنسبة للصوت الحالي). يجب إذا في الشفرة المصدرية أن تعيد قراءة جدول الـ512 `float` (بإعادة استدعاء `FMOD_Channel_GetSpectrum` كلّ 25 مث)، ثم تقوم بتحديث رسمك ذي الشرائط.

أعد قراءة الفصل حول التحكّم في الوقت بالـSDL لكي تتذكّر كيفية عمل ذلك. لديك الخيار بين `GetTicks` و الـcallbacks. استعمل ما تراه أكثر سهولة لك.

إنشاء التدرّج اللوني

في البداية، يمكنك تحقيق الشرائط بلون موحّد. يمكنك إذا إنشاء مساحات. يجب إذا أن تكون هناك 512 مساحة : واحدة من أجل كلّ شريط. كلّ مساحة تأخذ إذا بيكسلا واحدا كعُرض. و يختلف علوّ الشرائط بدلالة شدّة كلّ تردد.

أنصحك بعدها أن تقوم بتحسين : يجب على الشريط أن يميل للأحمر كلّما زادت كثافة الصوت. أي أنه على الشريط أن يكون أخضراً من الأسفل و أحمراً من الأعلى.

؟

لكن ... المساحة الواحدة لا يمكنها أن تأخذ سوى لونٍ واحدٍ لو عندما نستعمل الدالة `SDL_FillRect`. لا يمكننا إنشاء تدرّج لوني !

في الواقع، يمكننا بالتأكيد إنشاء مساحات بعَرْض 1 بيكسل و عُلو 1 بيكسل من أجل كلّ لون في التدرّج. لكن هذا سيأخذ بنا إلى إنشاء مساحات عديدة و لن يكون التحكّم فيها مثالياً !

كيف يمكن لنا أن نرسم بيكسلا ببيكسل ؟
لم أعلّمك هذا من قبل، لأنّ هذه التقنية لا تستحقّ فصلاً كاملاً. ستجد أنها في الواقع ليست صعبة.

في الواقع، لا تقترح الـSDL أية دالة للرسم بيكسلا ببيكسل. لكن لنا الحق في أن نكتبها بأنفسنا. لكي نقوم بهذا، يجب إتّباع هذه الخطوات النموذجية بالترتيب :

1. استدع الدالة `SDL_LockSurface` لنعلن للـSDL أننا سنقوم بالتعديل على المساحة يدوياً. هذا "يعطّل" المساحة للـSDL و ستكون وحدك قادراً على التحكّم فيها مادامت المساحة معطّلة.

هنا، أنصحك بأن تعمل بمساحة واحدة فقط : الشاشة. إذا أردت رسم بيكسل في منطقة محددة من الشاشة، يجب عليك تعطيل المساحة `screen` :

```
SDL_LockSurface(screen);
```

2. يمكنك بعد ذلك تغيير محتوى كلّ بيكسل من المساحة. بما أن الـSDL لا تقترح أية دالة للقيام بهذا، يجب أن نكتبها بأنفسنا في البرنامج.

سأعطيك هذه الدالة، و التي استخرجتها من الملفات التوجيهية للـSDL. هي معقّدة أكثر لأنها تعمل على المساحة مباشرة و تحكم في كلّ أعماق اللون الممكنة (بيتات على البيكسل). لا تحتاج لحفظها أو فهمها، قم بنسخها ببساطة في البرنامج لكي تتمكّن من استعمالها :

```
void setPixel(SDL_Surface *surface, int x, int y, Uint32 pixel)
{
        int bpp = surface->format->BytesPerPixel;

        Uint8 *p = (Uint8 *)surface->pixels + y * surface->pitch + x * bpp
            ;

        switch(bpp) {
                case 1:
                *p = pixel;
                break;

                case 2:
                *(Uint16 *)p = pixel;
                break;

                case 3:
```

```
17                if(SDL_BYTEORDER == SDL_BIG_ENDIAN) {
18                        p[0] = (pixel >> 16) & 0xff;
19                        p[1] = (pixel >> 8) & 0xff;
20                        p[2] = pixel & 0xff;
21                } else {
22                        p[0] = pixel & 0xff;
23                        p[1] = (pixel >> 8) & 0xff;
24                        p[2] = (pixel >> 16) & 0xff;
25                }
26                break;
27
28                case 4:
29                *(Uint32 *)p = pixel;
30                break;
31        }
32 }
```

هي سهلة الاستعمال. ابعث لها المعاملات التالية :

- المؤشّر نحو المساحة التي تريد التعديل عليها (يجب أن تكون معطّلة بواسطة `SDL_LockSurface`).
- وضعية الفاصلة الخاصة بالبيكسل الذي نريد التعديل عليه في المساحة (`x`).
- وضعية الترتيبة الخاصة بالبيكسل الذي نريد التعديل عليه في المساحة (`y`).
- اللون الجديد الذي نعطيه للبيكسل. يجب أن يكون هذا اللون بصيغة `Uint32`. يمكنك إذا توليده بالاستعانة بالدالة `SDL_MapRGB` التي تتقنها جيداً الآن.
- أخيراً، حينما تنتهي من العمل على المساحة، يجب ألا تنسى أن تزيل تعطيلها باستدعاء `SDL_UnlockSurface`.

```
1 SDL_UnlockSurface(screen);
```

شفرة ملخّصة للمثال

لو نلخّص، ستجد بأن كلّ شيء سهل.
هذه الشفرة ترسم بيكسلا أحمرا في منتصف المساحة `screen` (أي في منتصف النافذة).

```
1 SDL_LockSurface(screen); // We lock the surface
2 setPixel(screen, screen->w / 2, screen->h / 2, SDL_MapRGB(screen->format, 255,
     0, 0)); // We draw a red pixel in the middle of the screen
3 SDL_UnlockSurface(screen); // We unlock the surface
```

من هذه القاعدة، يجدر بك أن تتمكن من تحقيق التدرّج اللوني من الأخضر للأحمر (يجب أن تستعمل الحلقات التكرارية).

2.28 التصحيح

إذا، كيف وجدت الموضوع ؟ ليس صعب الفهم، يجب فقط القيام ببعض الحسابات، خاصة من أجل تحقيق التدرّج اللوني. مستوى التمرين هو مستوٍ عام، يجب فقط أن تفكّر أكثر.

بعض الأشخاص يأخذون وقتاً أطول من آخرين لإيجاد التصحيح. إذا لم تتمكّن من حلّ التمرين، هذا ليس سيئاً. ما يهمّ هو أن ننتهي بالوصول إلى هدفنا. مهما كان المشروع الذي تعمل عليه، فسيكون هناك بالتأكيد أوقات نجد فيها أنه لا ينقصنا أن نجيد البرمجة لكي نتمكّن من حلّ المشكل، يجب أيضاً أن نكون منطقيين و نجيد التفكير.

سأعطيك الشفرة المصدرية الكاملة. لقد علّقت عليها بشكل كافٍ :

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <SDL/SDL.h>
4  #include <fmodex/fmod.h>
5  #define WINDOW_WIDTH 512 // MUST stay equal to 512 because there are 512 bars
       corresponding to 512 floats
6  #define WINDOW_HIGHT 400 // You can change this.
7  #define RATIO (WINDOW_HIGHT / 255.0)
8  #define LIMIT_TIME_TO_REFRESH 25 // Time in ms between two updates of the graph
        (25 is the minimum)
9  #define SPECTERUM_SIZE 512
10 void setPixel(SDL_Surface *surface, int x, int y, Uint32 pixel);
11 int main(int argc, char *argv[])
12 {
13         SDL_Surface *screen = NULL;
14         SDL_Event event;
15         int cont = 1, barHeight = 0, currentTime = 0, previousTime = 0, i = 0,
               j = 0;
16         float spectrum[SPECTERUM_SIZE];
17         /* Initializing FMOD:
18         Load FMOD, the music and start playing the music
19         */
20 
21         FMOD_SYSTEM *system;
22         FMOD_SOUND *music;
23         FMOD_CHANNEL *channel;
24         FMOD_RESULT result;
25         FMOD_System_Create(&system);
26         FMOD_System_Init(system, 1, FMOD_INIT_NORMAL, NULL);
27         // We open the music
28         result = FMOD_System_CreateSound(system, "hype_home.mp3", FMOD_SOFTWARE
                | FMOD_2D | FMOD_CREATESTREAM, 0, &music);
29         // We check if it has been opened correctly (IMPORTANT)
30         if (result != FMOD_OK)
31         {
32                 fprintf(stderr, "Can't read the mp3 file\n");
```

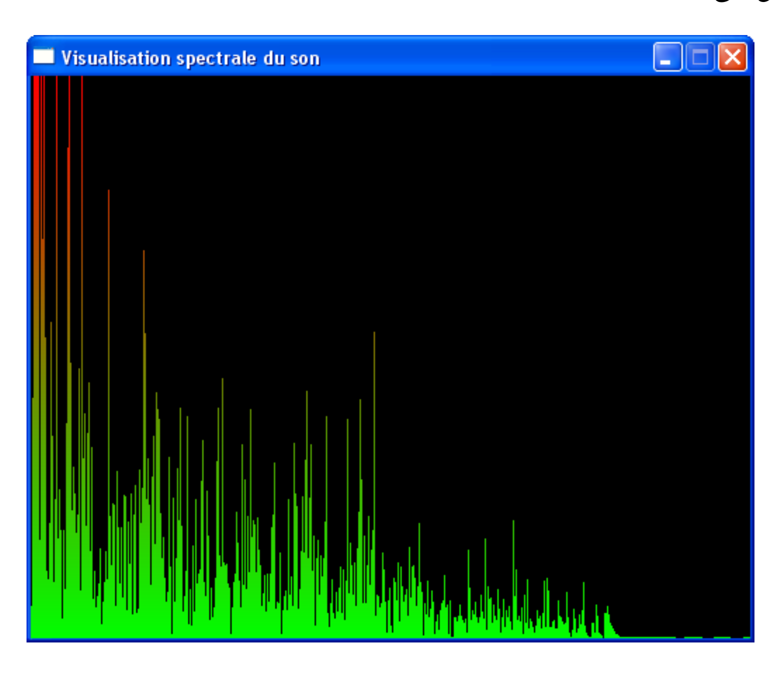
```
33              exit(EXIT_FAILURE);
34          }
35          // We play the music
36          FMOD_System_PlaySound(system, FMOD_CHANNEL_FREE, music, 0, NULL);
37
38          // We get the channel pointer
39          FMOD_System_GetChannel(system, 0, &channel);
40          /*
41          Initializing the SDL:
42          ---------------------
43          We load the SDL, open a window and write in its title bar.
44          We get also a pointer to the surface screen which will be the only
                surface to use in this program
45          */
46          SDL_Init(SDL_INIT_VIDEO);
47          screen = SDL_SetVideoMode(WINDOW_WIDTH, WINDOW_HIGHT, 32, SDL_SWSURFACE
                 | SDL_DOUBLEBUF);
48          SDL_WM_SetCaption("Visualisation spectrale du son", NULL);
49          // Main loop
50          while (cont)
51          {
52                  SDL_PollEvent(&event); // We have to use PollEvent because we
                        don't have to wait for the user's event to refresh the
                        window
53                  switch(event.type)
54                  {
55                          case SDL_QUIT:
56                          cont = 0;
57                          break;
58                  }
59                  // We clear the screen every time before drawing the graph (
                        black wallpaper)
60                  SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 0, 0, 0))
                        ;
61                  /* Managing the time
62                  -----------------
63                  We compare between the current time and the previous one (the
                        last iteration of the loop).
64                  If the difference is less than 25 ms (updating time limit)
65                  Then we wait until 25ms pass.
66                  After that, we update previousTime with the new time. */
67                  currentTime = SDL_GetTicks();
68                  if (currentTime - previousTime < LIMIT_TIME_TO_REFRESH)
69                  {
70                          SDL_Delay(LIMIT_TIME_TO_REFRESH-(currentTime-
                                previousTime));
71                  }
72                  previousTime = SDL_GetTicks();
73                  /* Drawing the sound spectrum
74                  ------------------------
```

```
75              It's the most important part. We have to think a little bit
                   before drawing the spectrum. Maybe it's hard but it's
                   possible, here is the proof.
76              We fill the 512 floats table via FMOD_Channel_GetSpectrum()
77              Then we work pixel by pixel on the surface screen to draw the
                   bars.
78              We make a first loop to browse the window in width.
79              The second loop browses the window in height to draw the bars.
80              */
81
82              /* We fill the 512 floats table. I've chosen to be interested
                   in the left output */
83              FMOD_Channel_GetSpectrum(channel, spectrum, SPECTERUM_SIZE, 0,
                   FMOD_DSP_FFT_WINDOW_RECT);
84              SDL_LockSurface(screen);
85              /* We block the surface screen because we're going to directly
                   modify its pixels */
86
87              /* LOOP 1 : We browse the window in width (for every vertical
                   bar) */
88              for (i = 0 ; i < WINDOW_WIDTH ; i++)
89              {
90                      /* We calculate the vertical bar's height that we're
                           going to draw.
91                      spectrum[i] will return a number between 0 and 1 that
                           we're going to multiply by 20 to zoom in order to
                           have a better view (As I said).
92
93                      The, we multiply by WINDOW_HEIGHT so the bar will be
                           expanded comparing to the window's size. */
94
95                      barHeight = spectrum[i] * 20 * WINDOW_HIGHT;
96                      /* We verify that the bar doesn't exceed the height of
                           the window
97                      If it's the case, we crop the bar so it become equal to
                            the window's height. */
98                      if (barHeight > WINDOW_HIGHT)
99                              barHeight = WINDOW_HIGHT;
100                     /* LOOP 2 : we browse in height the vertical bar to
                           draw it */
101
102                     for (j = WINDOW_HIGHT - barHeight ; j < WINDOW_HIGHT ;
                           j++)
103                     {
104                             /* We draw each pixel of the bar with the right
                                    colour.
105                             We simply vary the red and green colours, each
                                   one in a different way.
106
107                             j doesn't vary between 0 and 255 but between 0
```

```
                                    and WINDOW_HEIGHT.
108
109                             If we want to adapt it proportionally to the
                                    window's height, we can simply calculate j
                                    / RATIO, where RATIO is equal to (
                                    WINDOW_HEIGHT / 255.0).
110
111                             It tooks for me 2-3 minutes so I can find the
                                    write calculation to do, every one can do
                                    it. You just have to think a little bit */
112
113                             setPixel(screen, i, j, SDL_MapRGB(screen->
                                    format, 255 - (j / RATIO), j / RATIO, 0));
114                     }
115             }
116             SDL_UnlockSurface(screen); /* We have finished working on the
                    screen, we block the surface */
117             SDL_Flip(screen);
118         }
119         /* The program is finished.
120         We free the music from the memory
121         And we close FMOD and SDL */
122
123         FMOD_Sound_Release(music);
124         FMOD_System_Close(system);
125         FMOD_System_Release(system);
126         SDL_Quit();
127         return EXIT_SUCCESS;
128 }
129 /* The function setPixel lets us draw a surface pixel by pixel */
130
131 void setPixel(SDL_Surface *surface, int x, int y, Uint32 pixel)
132 {
133         int bpp = surface->format->BytesPerPixel;
134         Uint8 *p = (Uint8 *)surface->pixels + y * surface->pitch + x * bpp;
135         switch(bpp)
136         {
137                 case 1:
138                 *p = pixel;
139                 break;
140                 case 2:
141                 *(Uint16 *)p = pixel;
142                 break;
143                 case 3:
144                 if(SDL_BYTEORDER == SDL_BIG_ENDIAN)
145                 {
146                         p[0] = (pixel >> 16) & 0xff;
147                         p[1] = (pixel >> 8) & 0xff;
148                         p[2] = pixel & 0xff;
149                 }
```

```
150                 else
151                 {
152                         p[0] = pixel & 0xff;
153                         p[1] = (pixel >> 8) & 0xff;
154                         p[2] = (pixel >> 16) & 0xff;
155                 }
156                 break;
157                 case 4:
158                 *(Uint32 *)p = pixel;
159                 break;
160         }
161 }
```

يجدر بك أن تحصّل على نتيجة تشبه الصورة التالية :



من المعلوم أن النتيجة المرئية أفضل لتقدير النتيجة. أنصحك بالاطلاع عليها من هنا :

https://openclassrooms.com/uploads/fr/ftp/mateo21/spectre.html (4.3 Mo)

لاحظ أن ضغط الملف أنقص من جودة الصوت و عدد الصور في الثانية.

الأفضل هو أن تقوم بتحميل البرنامج كاملاً (مرفقاً بالشفرة المصدرية) لكي تجربه عندك. يمكنك حينها تقدير البرنامج في ظروف أفضل.

تنزيل الملف التنفيذي و الشفرة المصدرية الكاملة :

https://openclassrooms.com/uploads/fr/ftp/mateo21/spectre.zip

X

يجب قطعاً أن يكون الملف `Hype_Home.mp3` متواجداً في مجلّد المشروع لكي يشتغل البرنامج (و إلا فسيتوقف حالاً).

أفكار للتحسين

يمكن دائما تحسين البرنامج. هنا، لدي مثلاً أفكار تمديد كثيرة يمكنها أن تصل بك إلى إنشاء برنامج صغير لقراءة الملفات MP3.

- سيكون من الجيد أن نختار بأنفسنا الملف MP3 الذي نريد قراءته. يمكن مثلاً أن نقدّم لائحة تضم كلّ الملفات بذات الصيغة و المتواجدة في مجلّد المشروع. لم نرَ كيف نقوم بذلك، لكن يمكنك وحدك أن تكتشف ذلك. كمساعدة : استعمل المكتبة dirent (قم بتضمين الملف `dirent.h`). عليك بالبحث في الأنترنت لتعرف كيفيّة العمل بها.
- إذا كان البرنامج قادرا على التحكّم في لائحات التشغيل (Playlists) المشغّلة، سيكون أمراً أفضل. توجد كثير من صيغ اللائحات و أشهرها الصيغة M3U.
- يمكنك إظهار اسم الـMP3 التي أنت بصدد تشغيله في النافذة مثلاً (يجب استعمال SDL_ttf).
- يمكنك إظهار مؤشّر يشير إلى المكان في القراءة الّذي وصل إليه التشغيل، هذا ما يفعله أغلب قارئي الـMP3.
- يمكنك أيضاً أن تقترح التعديل على قوة الصوت.
- إلى آخره.

باختصار، هناك الكثير لفعله. لديك إمكانيّة إنشاء قارئي أصوات ممتازة، ليس عليك سوى كتابة الشفرة الخاصة بها !

الجزء د

هياكل البيانات

الفصل 29

القوائم المتسلسلة (Linked lists)

لكي نخزّن المعلومات في الذاكرة، استعملنا متغيّرات بسيطة (من نوع `int`، `double` ...)، كما استعملنا جداول و هياكل مخصّصة. إذا أردت تخزين سلسلة من البيانات، فالأبسط غالباً هو استعمال جداول.

لكن تصبح الجداول أحياناً محدودة جداً. مثلاً، إذا أنشأت جدولاً ذو 10 خانات ثم تبيّن لك لاحقاً في البرنامج أنك تحتاج إلى حجم أكبر، سيكون من المستحيل تكبير حجم الجدول. و أيضاً لا يمكنك إدخال خانة إلى وسط الجدول.

تمثّل القوائم المتسلسلة طريقة لتنظيم البيانات في الذاكرة بطريقة أكثر مرونة. و بما أن لغة الـC لا تقترح قاعدياً هذا النظام من التخزين، سيكون علينا أن ننشئه بأنفسنا. سيكون تمريناً ممتازاً يساعدك على أن ترتاح أكثر مع هذه اللغة.

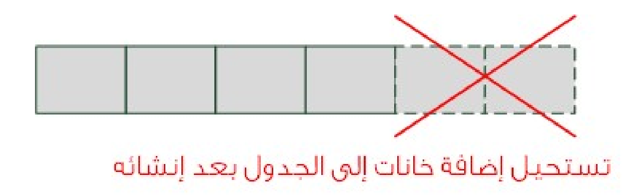
1.29 تمثيل قائمة متسلسلة

ماهي القائمة المتسلسلة ؟ أقترح عليك أن تنطلق من نموذج الجدول. يمكن تمثيل الجدول في الذاكرة بالطريقة التي توضّحها الصورة التالية. نتكلّم هنا عن جدول يحتوي على خانات من نوع `int`.



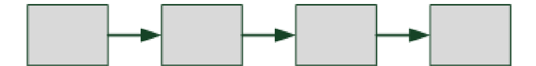
> اخترت هنا تمثيل الجدول أفقياً، لكن يمكن تمثيله عمودياً كذلك، هذا لا يهم.

كما قلت لك في المقدّمة، مشكل الجداول يكمن في كونها ثابتة. لا يمكن تكبير حجمها، إلا إذا فكّرنا في إعادة إنشائها من جديد و تكون أكبر (لاحظ الشكل التالي). أيضاً، لا يمكن أن نضيف عنصراً في وسط الجدول إلا إذا قمنا بإزاحة كلّ العناصر الأخرى.

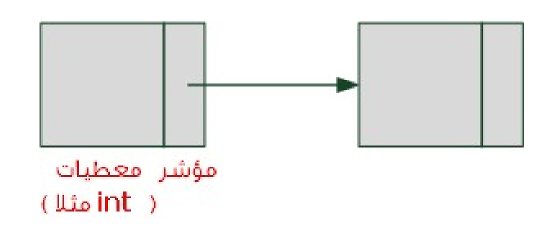


لا تقترح علينا لغة الـC نظاماً آخراً لتخزين البيانات، لكن من الممكن أن ننشئ بأنفسنا هذا النظام بعناصره الكاملة : ستكون الغاية من هذا الفصل و الفصول الموالية اكتشاف حلول لهذا المشكل.

القائمة المتسلسلة هي طريقة لتنظيم سلسلة من البيانات في الذاكرة. هذا يسمح بجمع هياكل (structures) مرتبطة ببعضها البعض بواسطة مؤشّرات. يمكننا تمثيلها كالتالي :



يمكن لكلّ عنصر أن يحتوي على ما نريد : قيمة من نوع `int` أو أكثر، `double` ... بالإضافة إلى ذلك، كلّ عنصر يحتوي على مؤشّر نحو العنصر الموالي :



أعرف بأن كلّ هذه المعلومات نظرية و ربّما تبدو لك غير واضحة الآن. احفظ فقط طريقة اتّصال العناصر ببعضها : هي تشكل **سلسلة من المؤشّرات**، و من هنا نجد الاسم "قائمة متسلسلة".

> على عكس الجداول، لا تتموضع عناصر القائمة المتسلسلة جنباً إلى جنب في الذاكرة. كل خانة تؤشّر نحو خانة أخرى لا تتواجد ضرورياً بجنب الأخرى.

2.29 بناء قائمة متسلسلة

فلنمرّ الآن إلى صلب الموضوع. سنحاول أن ننشئ بُنية تعمل بنفس المبدأ الذي اكتشفناه الآن.
أذكّرك بأن كلّ ما سنقوم به هنا يستدعي تقنيات لغة الـC التي تعرفها من قبل. لا يوجد شيء جديد، سنكتفي بإنشاء هياكلنا الخاصة و دوال ثم تحويلها إلى نظام منطقي قادر على العمل لوحده.

عنصر من القائمة

من أجل الأمثلة، سننشئ قائمة متسلسلة من أعداد صحيحة. كل عنصر من القائمة له شكل الهيكل التالي :

```
typedef struct Element Element;
struct Element
{
        int number;
        Element *next;
};
```

> يمكننا أيضاً إنشاء قوائم متسلسلة تحتوي أعداداً عشرية أو حتى جداول أو هياكل. مبدأ القوائم المتسلسلة صالح من أجل أي نوع من البيانات مهما كان، لكن هنا، أنصحك بتبسيط العملية حتى تفهم المبدأ.

قُمنا الآن بإنشاء عنصر واحد من القائمة، يوافق الصورة التي رأيناها أعلاه. على ماذا يحتوي الهيكل ؟

- قطعة بيانات، هنا تمثل في عدد من نوع `int` : يمكننا تغيير هذا بأي قطعة أخرى (`double`، جدول، ...). هذا يعتمد على نوع البيانات التي تريد تخزينها، سيكون عليك تغييرها على حسب حاجتك في البرنامج.

> إذا أردنا العمل بطريقة عامة، الأمثل هو استعمال مؤشّر نحو الفراغ : `void*`. هذا يسمح بالتأشير على أي نوع من البيانات.

- مؤشّر نحو عنصر من نفس النوع يسمّى `next`. هذا ما يسمح بوصل العناصر الواحد بالآخر : كلّ عنصر "يعلم" أين يتواجد العنصر الذي يليه في الذاكرة. كما قلتُ لك مسبقاً، الخانات لا تتواجد جنباً إلى جنب في الذاكرة. هذا هو الاختلاف الكبير بالنسبة للجداول. هذا ما سيمنحنا مرونة أكثر لأنه بإمكاننا بسهولة إضافة خانات أخرى لاحقاً حينما نحتاج إليها.

> و بالمقابل، لا يمكننا معرفة العنصر السابق، أي أنه من المستحيل الرجوع إلى الخلف انطلاقاً من عنصر من هذا النوع من القوائم. لأننا هنا نتكلّم عن قائمة "بسيطة التسلسل"، بينما توجد قوائم أخرى تسمّى "مزدوجة التسلسل" و تحتوي على مؤشّرات في كلتا الجهتين و بهذا فهي أصعب بقليل.

هيكل التحكّم

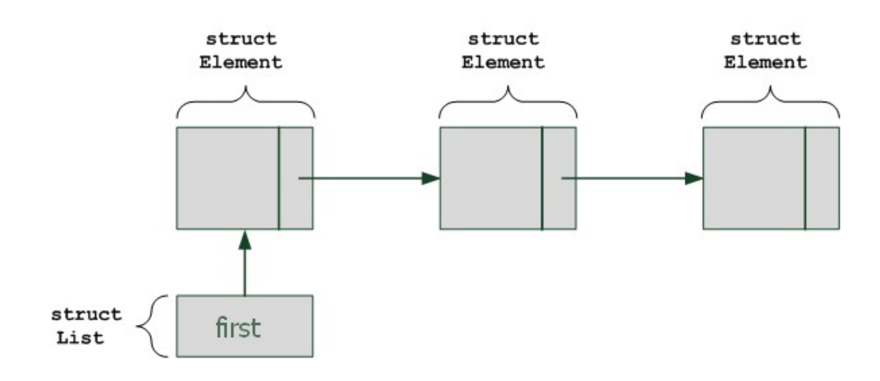
بالإضافة إلى الهيكل الذي نحن بصدد بنائه (و الذي نضاعفه بعدد المرات التي فيها عناصر أخرى)، سنحتاج إلى هيكل آخر لكي نتحكّم في كامل القائمة المتسلسلة. سيكون لهذا الهيكل الشكل التالي :

```c
typedef struct List List;
struct List
{
        Element *first;
};
```

هذا الهيكل `List` يحتوي على مؤشّر نحو أوّل عنصر من القائمة. في الواقع، يجب الاحتفاظ بعنوان العنصر الأول لكي نعرف أين تبدأ القائمة. إذا عرفنا العنصر الأول، يمكننا أن نجد العناصر الأخرى بـ"القفز" من عنصر لآخر بالإستعانة بالمؤشرات الموالية.

> م
> هيكل مكوّن من مركّب واحد هو في الغالب غير مفيد. و مع ذلك، أعتقد أننا سنحتاج أن نضيف إليه لاحقاً مركّبات أخرى، يمكننا مثلاً أن نخزّن به حجم القائمة، أي عدد العناصر التي تحتويها.

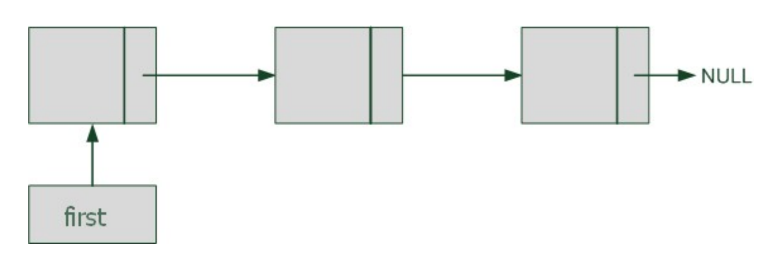
لن يكون علينا إنشاء سوى نسخة واحدة من الهيكل `List`. هي تسمح بالتحكّم في كلّ القائمة المتسلسلة :



آخر عنصر في القائمة

المخطط أصبح تقريباً كاملاً. ينقصه شيء أخير : نفضّل أن نحفظ العنصر الأخير من القائمة. في الواقع، يجب أن نتوقّف من التقدّم في القائمة المتسلسلة في لحظة ما. كيف سيتسنى لنا أن نقول للبرنامج : "توقف، هذا هو آخر عنصر" ؟

سيكون ممكناً أن نضيف إلى الهيكل `List` مؤشّرا نحو آخر عنصر. لكن هناك ما هو أبسط : يكفي أن يؤشّر آخر عنصر من القائمة على `NULL`، أي إعطاء المؤشّر `next` القيمة `NULL`. هذا سيسمح لنا أخيراً برسم مخطط كامل لبُنية القائمة المتسلسلة :



3.29 دوال معالجة القوائم المتسلسلة

لقد قُمنا بإنشاء هيكلين يسمحان لنا بالتعامل مع القوائم المتسلسلة :

- `Element`، الذي يوافق عنصراً من القائمة و الذي يمكن لنا أن نكرره بقدر المرات التي نريد.
- `List`، الذي يتحكّم في مجموع القائمة المتسلسلة. لن نحتاج إلا لنسخة واحدة منه.

هذا جيد، لكن ينقص الآن الأهم : الدوال التي ستتعامل مع القائمة المتسلسلة. في الواقع، لن نغيّر "يدويا" محتوى الهياكل في كلّ مرة نحتاج فيها إلى ذلك ! سيكون من الأكثر حكمة و الأكثر نظافة أن نمرّ بدوال تقوم بجعل العمل يتم بشكل تلقائيّ. يجب إنشاؤها هي بدورها.

كنظرة أولى، أقول بأنك نحتاج إلى دوال لكي :

- تهيّئ القائمة.
- تضيف عنصراً إليها.
- تحذف عنصراً منها.
- تُظهر محتواها.
- تحذف القائمة بأكملها.

يمكننا إنشاء دوال أخرى (حساب حجم القائمة مثلاً) لكن يمكن الاستغناء عنها. سنركّز الآن على الدوال التي قمتُ الآن بتعدادها، هذا سيُعطينا قاعدة جيدة. سأدعوك بعد ذلك إلى تحقيق دوال أخرى لكي تتدرّب بعدما تكون قد فهمت المبدأ جيّداً.

تهيئة القائمة

دالة تهيئة القائمة هي أول دالة سنحتاج إلى استدعائها. إذ أنها تقوم بإنشاء هيكل التحكّم و أوّل عنصر من القائمة. أقترح عليك الدالة أسفله و التي سنعلّق عليها بعد ذلك :

```
List *initialization()
{
        List *list = malloc(sizeof(*list));
        Element *element = malloc(sizeof(*element));
        if (list == NULL || element == NULL)
        {
                exit(EXIT_FAILURE);
        }
        element->number = 0;
        element->next = NULL;
        list->first = element;
        return list;
}
```

نبدأ بإنشاء هيكل التحكّم `list`.

لاحظ أن نوع البيانات هو `List` و أن المتغير يسمى `list`. تسمح طريقة كتابة الحرف الأول بالتفريق بينهما.

نحجز بطريقة حيّة هيكل التحكّم باستعمال `malloc`. الحجم الذي نحجزه محسوبٌ تلقائياً باستعمال `sizeof(*list)`. سيعرف الجهاز بأنه سيحجز المكان الكافي لتخزين الهيكل `List`.

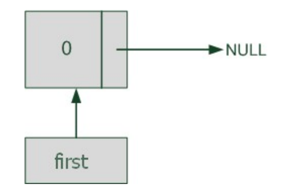
> كان بإمكاننا أن نكتُب أيضاً `sizeof(List)`، لكن إن أردنا لاحقاً تغيير نوع المؤشّر `list` سيكون علينا تحديث الـ `sizeof` كذلك.

نحجز أيضاً بنفس الطريقة الذاكرة اللازمة لتخزين أول عنصر. نتأكد من أن الحجز الحيّ قد تمّ بنجاح. في حالة خطأ، نوقف البرنامج حالاً باستدعاء الدالة `exit`.

أما إن تمّ كلّ شيء على ما يُرام، نعرّف قيم أول عنصر من القائمة المتسلسلة :

- يتم إعطاء 0 للمتغير `number` افتراضياً.
- المؤشّر `next` يؤشّر نحو `NULL` لأن أول عنصر في القائمة هو أيضاً آخر واحد لحدّ الآن. كما رأينا سابقاً، يجب على آخر عنصر أن يؤشّر نحو `NULL` ليشير إلى نهاية القائمة.

لقد نجحنا الآن في إنشاء قائمة في الذاكرة متكوّنة من عنصر واحد و لها الشكل التالي :

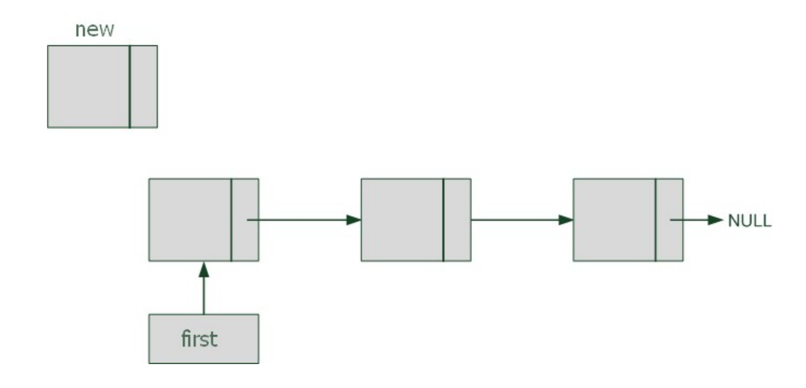


إضافة عنصر

هنا، تبدأ الأمور في التعقد قليلاً. أين سنضيف عنصراَ جديداً ؟ في بداية القائمة، في نهايتها أو في الوسط ؟

الإجابة هي أنه لدينا الخيار. سنكون أحراراً في اختيار ما نريد. في هذا الفصل، أقترح عليك أن تتعلّم كيفية إضافة عنصر إلى بداية القائمة. من ناحية، هذا الأمر سهل الفهم، و من ناحية أخرى سأعطيك الفرصة في نهاية الفصل في التفكير في طريقة إنشاء دالة تضيف عُنصراً في مكان محدد من القائمة.

يجدر بنا إنشاء دالة قادرة على أن تضيف عُنصراً جديداً إلى بداية القائمة. و لكي نفهم أكثر، تخيّل أننا في حالة مشابهة لما تعرضه الصورة الموالية : تتكون القائمة من ثلاثة عناصر و نريد أن نضيف لها عُنصراً جديدا في البداية :



يجب أن نقوم بتغيير وضعية المؤشّر `first` الخاص بالقائمة و أيضاً المؤشر `next` الخاص بالعنصر الجديد لكي "ندرج" هذا الأخير بشكل صحيح في القائمة. أقترح عليك هذه الشفرة المصدرية التي سنحللها لاحقاً :

```
1  void insertion(List *list, int newNumber)
2  {
3          // Creating a new element
4          Element *new = malloc(sizeof(*new));
5          if (list == NULL || new== NULL)
6          {
7                  exit(EXIT_FAILURE);
8          }
9          new->number = newNumber;
10         // Inserting the element at the beginning of the list
11         new->next = list->first;
12         list->first = new;
13 }
```

تأخذ الدالة `insertion` كمعاملات : عنصر التحكّم في القائمة (الذي يحتوي على عنوان أول عنصر) و العدد الذي نريد تخزينه في العنصر الجديد الذي سنقوم بإنشائه.

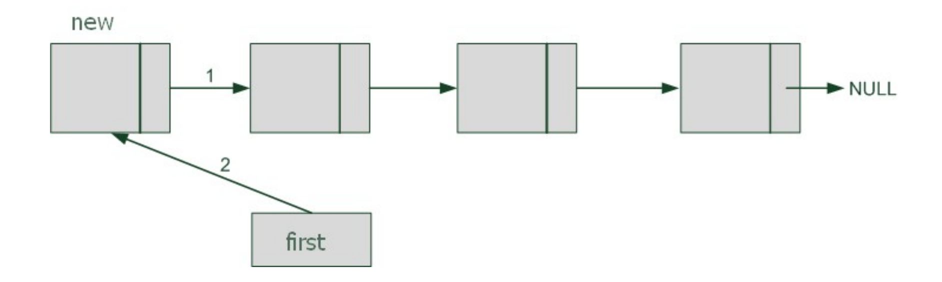
سنقوم أولا بحجز المكان اللازم لتخزين العنصر الجديد و نضع به العدد `newNumber`. تبقى إذا المرحلة الحساسة : إدراج العنصر الجديد في القائمة المتسلسلة.

لقد اخترنا هنا، تسهيلاً للعملية، إضافة العنصر إلى بداية القائمة. لكي نحدّث المؤشّرات بشكل صحيح، سنعتمد على الخطوتين التاليتين بهذا الترتيب المحدد :

1. تأشير العنصر الجديد نحو العنصر الذي سيليه مستقبلاً، أي العنصر الحالي الأول من القائمة.
2. تأشير المؤشّر `first` نحو العنصر الجديد.

! لا يمكننا القيام بهاتين الخطوتين في الترتيب المعاكس ! في الواقع، إن قمنا بجعل المؤشّر `first` يؤشّر أولا نحو العنصر الجديد، سنخسر عنوان العنصر الأول من القائمة ! جرّب ذلك، و ستفهم بعد ذلك لمَ عكس الخطوتين أمر مستحيل.

بإتباع الخطوتين سنتمكن من إدراج العنصر الجديد بشكل صحيح إلى القائمة المتسلسلة :



حذف عنصر

و نفس الشيء بالنسبة للإضافة، سنركّز الآن على عملية حذف أول عنصر من القائمة. تقنيا، يُسمح بمسح عنصر متواجد في وضعية محددة من وسط القائمة، سيكون هذا واحدا من التمارين التي أقترحها عليك في نهاية الفصل.

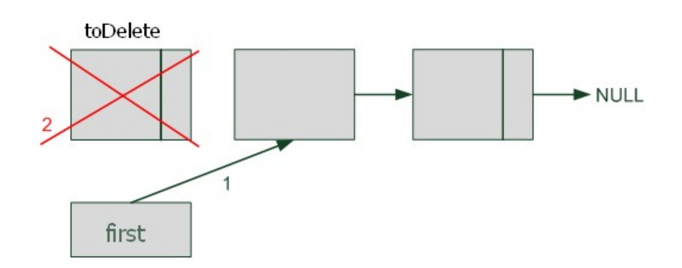
عملية حذف عنصر من القائمة المتسلسلة لا تطرح مشكلاً إضافياً. يجب فقط أن نجري التغييرات على المؤشّرات في الترتيب الصحيح لكي لا "نخسر" أية معلومة.

```c
void deletion(List *list)
{
        if (list == NULL)
        {
                exit(EXIT_FAILURE);
        }
        if (list->first != NULL)
        {
                Element *toDelete = list->first;
                list->first = list->first->next;
                free(toDelete);
        }
}
```

نبدأ بالتأكد من أن المؤشّر الذي استقبلناه لا يساوي `NULL`، و إلا فلن نتمكّن من العمل. نتأكد بعد ذلك إذا كان هناك على الأقل عنصر واحد في القائمة و إلا فلا يوجد أي شيء لنقوم به.

بعد الانتهاء من هذه الاختبارات، يمكننا حفظ عنوان العنصر الذي نريد حذفه في مؤشّر نسميه `toDelete`. يؤشّر بعد ذلك المؤشّر `first` نحو العنصر الجديد الأول، و الذي هو حالياً في الوضعية الثانية من القائمة المتسلسلة.

لا يبقى إلا تحرير العنصر الموافق للمؤشّر `toDelete` باستعمال الدالة `free` :



هذه الدالة قصيرة لكن هل يمكنك إعادة كتابتها لوحدك ؟ يجب أن نفهم جيداً بأننا يجب أن نقوم بالعمل اتّباعاً لخطوات محددة :

1. تأشير `first` نحو العنصر الثاني.
2. مسح العنصر الأول باستعمال `free`.

إذا قُمنا بالعكس، سنخسر عنوان العنصر الثاني !

إظهار محتوى القائمة المتسلسلة

لكي نرى بشكل واضح محتوى القائمة المتسلسلة، سيكون من الأمثل أن نكتب دالة عرض ! يكفي أن ننطلق من العنصر الأول و إظهار العناصر واحداً تلو الآخر بـ"القفز" من كتلة لأخرى.

```
1  void displayList(List *list)
2  {
3          if (list == NULL)
4          {
5                  exit(EXIT_FAILURE);
6          }
7          Element *current = list->first;
8          while (current != NULL)
9          {
10                 printf("%d -> ", current->number);
11                 current = current->next;
12         }
13         printf("NULL\n");
14 }
```

هذه الدالة بسيطة : ننطلق من العنصر الأول و نُظهر محتوى كلّ عنصر (عدد). نستفيد من المؤشّر `next` لننتقل إلى العنصر المُوالي في كلّ مرة.

يمكننا أن نستمتع بتجريب إنشاء قائمتنا المتسلسلة و إظهارها في الـ `main` :

```
1  int main()
2  {
3          List *myList = initialization();
4          insertion(myList, 4);
5          insertion(myList, 8);
6          insertion(myList, 15);
7          deletion(myList);
8          displayList(myList);
9          return 0;
10 }
```

بالإضافة إلى العنصر الأول (و الذي تركناه هنا يحمل القيمة 0)، نضيف ثلاثة عناصر جديدة لهذه القائمة. ثم نقوم بحذف عنصر واحد. في النهاية، يتم إظهار محتوى القائمة المتسلسلة بالشكل التالي :

```
8 -> 4 -> 0 -> NULL
```

4.29 اذهب بعيدا

لقد قُمنا الآن بكتابة الدوال اللازمة للتحكّم في قائمة متسلسلة : التهيئة، إضافة عنصر، حذف عنصر، إلخ. إليك بعض الدوال التي تنقص و التي أدعوك إلى كتابتها، سيكون هذا بمثابة تمرين جيد لك !

- **إضافة عنصر في وسط القائمة** : حالياً، لا يمكننا إضافة عناصر إلا في بداية القائمة، هذا كافٍ بشكل عام. أما إن أردنا إضافة عنصر إلى منتصف القائمة، سيكون علينا أن نكتب دالة تأخذ معاملا إضافيّا : عنوان العنصر الذي يسبق العنصر الجديد في القائمة. ستقوم الدالة بالتقدّم في القائمة إلى حين الوصول إلى العنصر المُراد و تقوم بإضافة العنصر الجديد بعده مباشرة.
- **حذف عنصر من وسط القائمة** : المبدأ نفسه بالنسبة للإضافة في وسط القائمة. هذه المرة، يجب عليك أن تضيف معاملا يمثل عنوان العنصر الذي نريد حذفه.
- **تدمير القائمة** : يكفي أن نقوم بحذف كل العناصر واحداً تلو الآخر !
- **حجم السلسلة** : تشير هذه الدالة إلى كم من عنصر تتكون القائمة المتسلسلة. الأمثل، و في عوض أن يتم حساب هذه القيمة في كلّ مرة، هو أن نضيف عددا صحيحا `nbOfElements` إلى الهيكل `List`. يكفي أن نزيد من قيمته في كلّ مرة نضيف فيها عنصراً جديداً للقائمة و أن ننقص من قيمته في كلّ مرة نحذف عنصراً منها.

أنصحك بجمع كلّ دوال معالجة القوائم المتسلسلة في ملفين `linked_list.c` و `linked_list.h` مثلاً. ستكون أوّل مكتبة تكتبها بنفسك ! يمكنك إعادة استعمالها في كلّ برامجك الأخرى التي تحتاج فيها إلى القوائم المتسلسلة.

يمكنك تنزيل مشروع القوائم المتسلسلة الذي يحتوي الدوال التي اكتشفناها سوياً. ستكون هذه بمثابة قاعدة جيدة لك.

http://www.siteduzero.com/uploads/fr/ftp/mateo21/c/listes_chainees.zip

ملخّص

- تشكّل القوائم المتسلسلة طريقة جديدة لتخزين البيانات في الذاكرة. هي أكثر مرونة من الجداول لأنها تمكّننا من إضافة و حذف "خانات" في أي لحظة نريد.
- لا تحتوي لغة الـC على نظام تحكّم في القوائم المتسلسلة، إذ يجب أن نكتبه بأنفسنا ! يعتبر هذا طريقة ممتازة للتقدّم في الخوارزميات و البرمجة بشكل عام.
- في قائمة متسلسلة، كل عنصر هو عبارة عن هيكل يحتوي عنوان العنصر الموالي.
- يُنصح بإنشاء هيكل تحكّم (من نوع `List` في حالتنا هذه) يحتوي عنوان أول عنصر في القائمة.
- توجد نسخة محسّنة - لكن أكثر تعقيداً - من القوائم المتسلسلة و نسمّيها "القوائم مزدوجة التسلسل"، و التي يحتوي كلّ عنصر فيها على عنوان العنصر السابق أيضاً.

الفصل 30

المكدّسات و الطوابير (Stacks and Queues)

لقد اكتشفنا مع القوائم المتسلسلة طريقة جديدة أكثر مرونة من الجداول لتخزين البيانات. هذه القوائم مرنة بشكل خاص لأنه يمكننا أن نُدرج فيها و نحذف منها بيانات من أي مكان أردنا و في أية لحظة.

المكدّسات و الطوابير التي سنكتشفها هنا هما شكلان خاصّان نوعاً ما من القوائم المتسلسلة. فهما تسمحان بالتحكّم بالطريقة التي تُضاف بها العناصر الجديدة إليها. هذه المرة لن نقوم بإضافة عناصر جديدة في وسط القائمة، بل فقط في البداية أو النهاية.

المكدّسات و الطوابير تعتبران مفيدتان للغاية من أجل البرامج التي تحلل المعطيات الّتي تصل بالتدريج. سنرى بالتفصيل كيف تعملان في هذا الفصل.

تتشابه المكدّسات و الطوابير كثيراً، لكنهما تختلفان اختلافاً بسيطاً ستتعرف عليه بسرعة. سنكتشف أولاً المكدّسات و التي ستذكّرك بالقوائم المتّصلة بشكل كبير.

بشكل عام، سيكون هذا الفصل بسيطاً إذا كنت قد فهمت جيّداً كيفية عمل القوائم المتسلسلة. إن لم تكن هذه حالتك، فأعد قراءة الفصل السابق لأننا سنحتاج إليه.

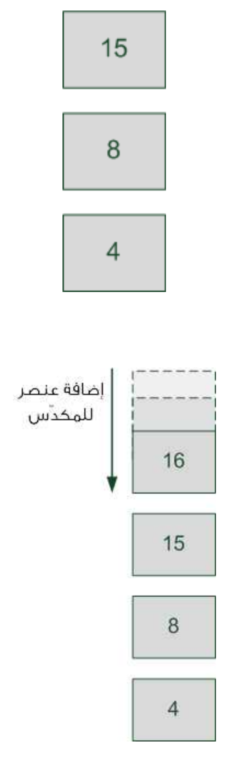
1.30 المكدّسات (Stacks)

تخيّل مكدّساً للقطع النقدية (الصورة التالية). يمكنك إضافة قطع أخرى واحدة تلو الأخرى في أعلى المكدّس، و يمكنك أيضاً نزع القطع من أعلى المكدّس. بالمقابل، لا يمكن نزع قطعة من أسفل المكدّس. إن أردت التجريب، أتمنى لك حظّاً موفّقاً !

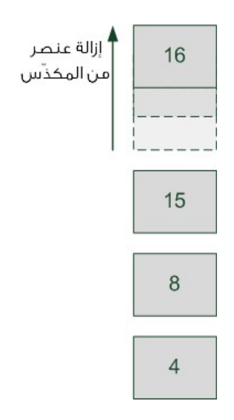


كيفية عمل المكدّسات

مبدأ عمل المكدّسات في البرمجة ينصّ على تخزين البيانات مع وصولها على التوالي واحدة فوق الأخرى لكي نستطيع استرجاعها فيما بعد. مثلا، تخيّل مكدّساً للأعداد الصحيحة من نوع `int` (الصورة الموالية). لو أضيف عنصراً (نتكلّم عن **التكديس**)، فستتم إضافته في أعلى المكدّس، تماماً كما في لعبة Tetris :

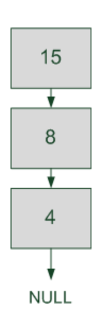


الأكثر أهمية هو وجود عملية تقوم باستخراج الأعداد من المكدّس. نحن نتكلّم عن **إلغاء التكديس**. نسترجع القيم واحدة تلو الأخرى، بدءً من الأخيرة الموضوعة أعلى المكدّس (الصورة الموالية). ننزع البيانات على التوالي حتى نصل إلى قاع المكدّس.



نسمي هذا بخوارزمية **LIFO**، و التي تعني "Last In First Out". الترجمة : "آخر عُنصر تمت إضافته، هو أول عنصر يخرج".

عناصر المكدّس مرتبطة ببعضها بطريقة القوائم المتسلسلة. فهي تحمل مؤشّراً نحو العنصر الموالي و لا تتموضع بالضرورة بجنب بعضها في الذاكرة. يجب على آخر عُنصر (في أقصى أسفل المكدّس) أن يؤشّر نحو `NULL` لكي يقول أنّنا ... لمسنا القاع :



؟

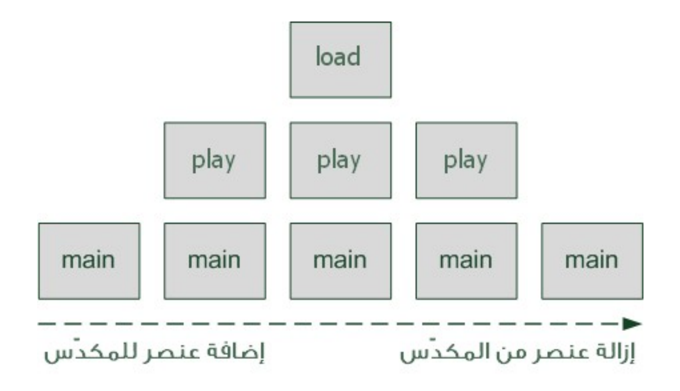
فيما ينفع كلّ هذا، واقعيّا ؟

توجد برامج تحتاج فيها إلى تخزين البيانات مؤقّتاً لإخراجها اعتماداً على ترتيب محدد : يجب على آخر عُنصر أدخلته أن يخرج هو الأول.

لأعطي مثالاً واقعياً، يستعمل نظام التشغيل في حاسوبك هذا النوع من الخوارزميات لكي يتذكر الترتيب الذي تم استدعاء الدوال فيه. تخيّل مثالا :

1. يبدأ البرنامج بالدالة `main` (مثل كل مرّة).
2. تستدعي فيها الدالة `play` .
3. تقوم هذه الدالة `play` بدورها باستدعاء الدالة `load` .
4. ما إن تنتهي الدالة `load` ، نعود إلى الدالة `play` .
5. ما إن تنتهي الدالة `play` ، نعود إلى الدالة `main` .
6. أخيراً، ما إن تنتهي الدالة `main` . لا تبقى أية دالة تحتاج إلى الاستدعاء، ينتهي البرنامج.

لكي "يتذكّر" الترتيب الذي تم فيه استدعاء الدوال، يُنشئ الحاسوب مكدّساً لهذه الدوال على التوالي :



هذا مثال واقعيّ عن استعمال المكدّسات. و بفضل هذه التقنية يعرف الجهاز الآن إلى أي دالة يجب عليه أن يعود. يمكنه أن يكدّس 100 استدعاء للدوال إن وجب الأمر، لكنّه سيرجع ليجد الدالة الرئيسية في أسفل المكدّس !

إنشاء نظام مكدّس

و الآن بما أننا فهمنا مبدأ عمل المكدّسات، فلنحاول بناء واحد. تماما مثل القوائم المتسلسلة، لا يوجد نظام مكدّس متضمّن في لغة الـC. يجب أن ننشئه بأنفسنا.

سيكون لكل عنصر من المكدّس هيكل مماثل للهيكل الخاص بالقائمة المتسلسلة :

```c
typedef struct Element Element;
struct Element
{
        int number;
        Element *next;
};
```

يحتوي هيكل التحكّم على عنوان أول عنصر من المكدس، أي العنصر المتواجد في الأعلى :

```c
typedef struct Stack Stack;
struct Stack
{
        Element *first;
};
```

سنحتاج ككل إلى الدوال التالية :

- تكديس عنصر.
- إلغاء تكديس عنصر.

ستلاحظ أنه، على خلاف القوائم المتسلسلة، لا نتكلّم لا عن الإضافة و لا عن الحذف. نتكلّم عن التكديس و إلغاء التكديس. لأن هاتين العمليتين محدودتان على عنصر واحد محدد، كما رأينا. بهذا، يمكننا إضافة و نزع عنصر من المكدّس من الأعلى فقط.

يمكننا أيضاً كتابة دالة لإظهار محتوى المكدّس، أمر عمليّ للتأكد من أن البرنامج يعمل بشكل صحيح.

هيا بنا !

التكديس (stacking)

يجدر بدالتنا `stack` أن تأخذ كعاملين هيكل التحكّم في المكدّس (من نوع `Stack`) و العدد الجديد لتخزينه.
أذكّرك بأننا نخزّن هنا أعداداً صحيحة `int`، لكن لا شيء يمنعنا من تعديل هذه الأمثلة لأنواع أخرى من البيانات. يمكننا تخزين أي شيء : أعداد `double`، محارف `char`، سلاسل محارف، جداول و حتى هياكل أخرى !

```
1  void stack(Stack *stk, int newNumber)
2  {
3          Element *new = malloc(sizeof(*new));
4          if (stk == NULL || new == NULL)
5          {
6                  exit(EXIT_FAILURE);
7          }
8          new->number = newNumber;
9          new->next = stk->first;
10         stk->first = new;
11 }
```

تتم الإضافة في بداية المكدّس لأنه، كما رأينا، يستحيل القيام بذلك في المنتصف. هذا مبدأ عمل المكدّسات، نضيف دائما من الأعلى.
بهذا، على عكس القوائم المتسلسلة، لا يجب أن ننشئ دالة لإدراج عنصر في منتصف المكدّس. يجب أن تكون الدالة `stack` هي الوحيدة الّتي يمكنها إضافة عناصر جديدة للمكدّس.

إلغاء التكديس (unstacking)

دور دالة إلغاء التكديس هو حذف العنصر المتواجد في أعلى المكدّس، قد تشك في ذلك. لكن يجب على هذه الدالة أيضا أن تُرجع إلينا العنصر الذي حذفته. في حالتنا، هو العدد الذّي كان موجودا في أعلى المكدّس.

هذه هي الطريقة التي نصل بها إلى عناصر المكدّس : ننزعها واحداً تلو الآخر. نحن لا نتقدّم فيها باحثين عن الوصول إلى ثاني و ثالث عنصر. بل نطلب دائمًا استرجاعاً على أول عنصر.

الدالة `unstack` ستُرجع إذا `int` يوافق العدد المتواجد في رأس المكدّس :

```
1  int unstack(Stack *stack)
2  {
3          if (stack == NULL)
4          {
5                  exit(EXIT_FAILURE);
6          }
7          int unstackedNumber = 0;
8          Element *unstackedElement = stack->first;
9          if (stack != NULL[1] && stack->first != NULL)
10         {
11                 unstackedNumber = unstackedElement->number;
```

```
12            stack->first = unstackedElement->next;
13            free(unstackedElement);
14        }
15        return unstackedNumber;
16 }
```

ملاحظة مُراجِع الكتاب[1]

يبدو لي أن مؤلّف الكتاب قد أضاف جزءً عديم الفائدة في الشرط الثاني : `stack != NULL`، فَبِوصول البرنامج إلى ذلك السطر سيكون هذا الجزء من الشرط خاطئا بكلّ تأكيد لأنّنا قد تحقّقنا من عكسه في الشرط الأوّل (`if(stack == NULL)`). و حتى عند عدم وجود الشرط الأوّل، فالتعليمة `stack->first` كانت ستعطّل البرنامج لأنّ مؤشّراً `NULL` لا يملك أيّة مركّبات !

نسترجع العدد الذي في رأس المكدّس لنبعثه في نهاية الدالة. نعدّل عنوان أول عنصر من المكدّس بما أن هذا الأخير يتغير. أخيراً، نحذف بالتأكيد رأس المكدّس القديم باستعمال `free`.

إظهار محتوى المكدّس

بالرغم من أن هذه الدالة غير ضروريّة (الدالتان `stack` و `unstack` كافيتان للتحكّم في المكدّس !)، ستكون مهمّة لاختبار عمل المكدّس و خاصّة "لمعاينة" النتيجة :

```
1  void displayStack(Stack *stack)
2  {
3      if (stack == NULL)
4      {
5          exit(EXIT_FAILURE);
6      }
7      Element *current = stack->first;
8      while (current != NULL)
9      {
10         printf("%d\n", current->number);
11         current = current ->next;
12     }
13     printf("\n");
14 }
```

بما أن هذه الدالة بسيطة بسخافة، فهي لا تحتاج شرحاً.

بالمقابل، حان الآن إذاً لكتابة الدالة الرئيسية لاختبار سلوك مكدّسنا :

```
1  int main()
2  {
3      Stack *myStack = initialization(); // See the previous chapter
4      stack(myStack, 4);
5      stack(myStack, 8);
6      stack(myStack, 15);
```

```
7         stack(myStack, 16);
8         stack(myStack, 23);
9         stack(myStack, 42);
10        printf("\nStack's state :\n");
11        displayStack(myStack);
12        printf("I unstack %d\n", unstack(myStack));
13        printf("I unstack %d\n", stack(myStack));
14        printf("\nStack's state :\n");
15        displayStack(myStack);
16        return 0;
17 }
```

نُظهر حالة المكدّس بعد الكثير من التكديس و مرة أخرى بعد كثير من إلغاء التكديس. نُظهر أيضاً العدد الذي قُمنا بحذفه في كلّ مرة نقوم بإلغاء التكديس. النتيجة في الكونسول هي التالية :

```
Stack's state :
42
23
16
15
8
4

I unstack 42
I unstack 23

Stack's state :
16
15
8
4
```

تأكّد أنك تفهم جيّداً ما يحصل في البرنامج. إذا فهمت هذا، فقد فهمت كيفية عمل المكدّسات !

يمكنك تنزيل مشروع المكدّسات كاملاً لو أردت :

http://www.siteduzero.com/uploads/fr/ftp/mateo21/c/piles.zip

2.30 الطوابير (Queues)

تشبه الطوابير المكدّسات كثيراً، إلا أنها تعمل بالإتجاه المعاكس !

كيفية عمل الطوابير

في هذا النظام، يتم وضع العناصر الواحد بعدَ الآخر. أوّل عنصر يخرج من الطابور هو أول عنصر يدخل إليه. نتكلّم هنا عن خوارزمية FIFO (First In First Out)، و هذا يعني : "أول من يصل هو أول من يخرج".

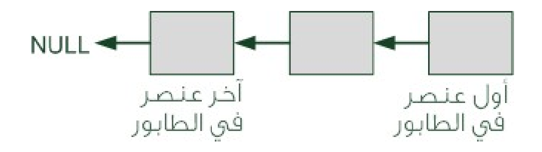
تسهل المشابهة بالحياة اليوميّة. حينما تشتري تذكرة لمشاهدة السينما، تقف في طابور شبّاك التذاكر (الصورة الموالية). باستثناء إن كنت أحد إخوة بائع التذاكر، يجدر بك الوقوف في الطابور و انتظار دورك مثل كلّ الآخرين. أول الواصلين هو أول من تتم خدمته.



في البرمجة، تفيد الطوابير في إيقاف مؤقّت للمعلومات حسب الترتيب الذي وصلت به. مثلاً، في برنامج مُحادثة، لو تتلقى ثلاثة رسائل يفصلها فارق زمني قصير جداً، يتم صفُّها في طابور واحدة تلو الأخرى في الذاكرة. ثم يتم إظهار أوّل رسالة وصلت ثم الثانية و هكذا.

يتم تخزين الأحداث التي تبعثها المكتبة SDL التي قُمنا بدراستها أيضا في طابور. إذا حرّكت الفأرة، يتم توليد حدث من أجل كلّ بيكسل تحرّك فوقه مؤشّر الفأرة. تخزن الـSDL الأحداث في طابور ثم تبعثها لنا واحدا واحدا في كلّ مرة نستدعي فيها الدالة `SDL_PollEvent` (أو `SDL_WaitEvent`).

في لغة الـC، الطابور هو قائمة متسلسلة أين يقوم كلّ عنصر فيها بالتأشير على العنصر الموالي، تماماً مثل المكدّسات. آخر عنصر من الطابور يؤشّر نحو `NULL` :



إنشاء نظام طابور

نظام الطابور يشبه ذلك الخاص بالمكدّسات. يوجد اختلاف بسيط في كون أن العناصر تخرج في الإتجاه المعاكس، لا يوجد شيء صعب إن كنت قد فهمت المكدّسات.

سننشئ هيكل `Element` و هيكل تحكّم `Queue` :

```c
typedef struct Element Element;
struct Element
{
        int number;
        Element *next;
};
typedef struct Queue Queue;
```

```c
struct Queue
{
        Element *first;
};
```

تماما كالمكدّسات، كل عنصر من الطابور سيكون من نوع `Element`. بالإستعانة بالمؤشّر `first` سنتوفّر دائماً على العنصر الأول و يمكننا من خلاله الصعود إلى آخر عنصر.

إضافة عنصر إلى الطابور (enqueuing)

الدالة التي تضيف عُنصُراً إلى الطابور تسمّى دالة "الإلحاق" (enqueuing). توجد حالتان :

- إما أن الطابور فارغ، في هذه الحالة يجب أن ننشئ الطابور بجعل المؤشّر `first` يؤشّر نحو العنصر الجديد الذي نحن بصدد انشائه.
- إما أن الطابور غير فارغ، في هذه الحالة يجب أن نتقدّم في الطابور إنطلاقاً من العنصر الأول حتى نصل إلى آخر عنصر. نضيف العنصر الجديد بعد آخر عنصر.

إليك ما يمكننا فعله عملياً :

```c
void enqueue(Queue *q, int newNumber)
{
        Element *new = malloc(sizeof(*new));
        if (q == NULL || new == NULL)
        {
                exit(EXIT_FAILURE);
        }
        new->number = newNumber;
        new->next = NULL;
        if (q->first != NULL) // The queue is not empty
        {
                // We move to the end of the queue
                Element *currentElement = q->first;
                while (currentElement->next != NULL)
                {
                        currentElement = currentElement->next;
                }
                currentElement->next = new;
        }
        else /* The queue is empty, it's our first element */
        {
                q->first = new;
        }
}
```

ترى في هذه الشفرة المصدرية تحليل كلتا الحالتين، كلّ منهما يجب أن تتم دراستها على حدى. الاختلاف مقارنةً بالمكدّسات، و الذي يضيف لمسة صعوبة صغيرة، هو أنه يجب التموضع في نهاية الطابور لوضع العنصر الجديد. لكن لا بأس فحلقة `while` كافية للقيام باللازم، هذا ما يمكنك ملاحظته.

إزالة عنصر من الطابور (dequeuing)

عملية إلغاء الإلحاق (dequeuing) تشابه كثيراً عملية إلغاء التكديس. بامتلاكنا مؤشّرا نحو أول عنصر من الطابور، يكفي أن ننزعه و أن نُرجع قيمته.

```c
int dequeue(Queue *queue)
{
        if (queue == NULL)
        {
                exit(EXIT_FAILURE);
        }
        int dequeuedNumber = 0;
        // We verify if there's something to dequeue
        if (queue->first != NULL)
        {
                Element *dequeuedElement = queue->first;
                dequeuedNumber = dequeuedElement->number;
                queue->first = dequeuedElement->next;
                free(dequeuedElement);
        }
        return dequeuedNumber;
}
```

حان دورك !

تبقى دالة إظهار محتوى الطابور `displayQueue` و عملها مشابه لما قمنا به مع المكدّسات. سيسمح لك هذا بالتأكد من سلامة عمل الطابور.

قم بعد ذلك بكتابة `main` من أجل تجريب البرنامج. يجدر بنتيجة البرنامج أن تشبه هذه :

```
Queue's state :
4 8 15 16 23 42

I dequeue 4
I dequeue 8

Queue's state :
15 16 23 42
```

يجدر أن تكون قادراً على إنشاء مكتبة الطوابير الخاصة بك، بملفات `queue.h` و `queue.c` مثلاً.

أقترح عليك تنزيل مشروع التحكّم في الطوابير كاملاً إن أردت. إنه يتضمّن الدالة `displayQueue` :

http://www.siteduzero.com/uploads/fr/ftp/mateo21/c/files.zip

ملخّص

- تسمح المكدّسات و الطوابير بتنظيم معطيات في الذاكرة عند وصولها بالتوالي.
- تستعمل المكدّسات و الطوابير نظام قوائم متسلسلة لتجميع العناصر.
- في حالة المكدّسات، تتم إضافة المعطيات الواحدة فوق الأخرى. و إن أردنا استخراج بيانة، فسنستخرج آخر واحدة و التي كنّا بصدد إضافتها (الأحدث). نتكلّم هنا عن خوارزمية LIFO (Last In First Out).
- في حالة الطوابير، تتم إضافة المعطيات الواحدة بعد الأخرى. نقوم باستخراج البيانة الأولى و التي قمنا بإضافتها أولا للطابور (الأقدم). نتكلّم عن خوارزمية FIFO (First In First Out).

الفصل 31

جداول التجزئة (Hash tables)

للقوائم المتسلسلة نقطة ضعف كبيرة في حال أردنا قراءة محتواها : يستحيل الوصول إلى عنصر معيّن مباشرة. يجب التقدّم في القائمة عنصراً بعنصر حتى نجد العنصر الذي نريد. هذا يطرح مشاكل من ناحية الأداء ما إن يكون حجم القائمة المتسلسلة ضخماً. تخيّل قائمة متسلسلة تتكوّن من 1000 عنصر بينما العنصر الذي نبحث عنه موجود في آخرها !

تمثّل جداول التجزئة طريقة أخرى لتخزين البيانات. حيث أنها تستند على مبدأ الجداول في لغة الـC و التي نعرف التعامل معها جيّداً. ماهي فائدتها الكُبرى ؟ هي تسمح بإيجاد سريع لعنصر محدد، سواء كان الجدول يحتوي 100، 1000، 10000 خانة أو حتى أكثر !

1.31 لماذا نستعمل جدول تجزئة ؟

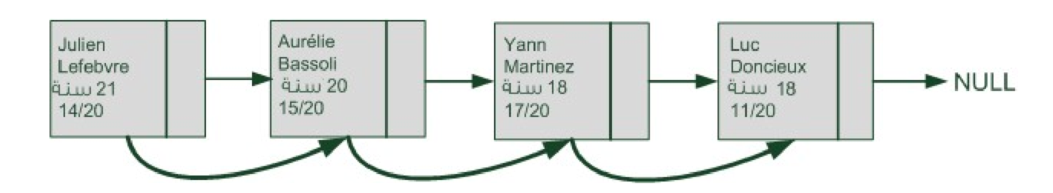
لننطلق من المشكل الذي تطرحه القوائم المتسلسلة. هذه الأخيرة مرنة بشكل خاص، هذا ما استطعنا ملاحظته : يمكننا إضافة أو إزالة خانات في أي لحظة نريد، بينما يكون الجدول "ثابتاً" ما إن يتم إنشاؤه.

لكن، كما قلتُ لك في المقدّمة، للقوائم المتسلسلة عيب كبير : إذا أردنا استرجاع عنصر محدد من القائمة، يجب تصفّح هذه الأخيرة حتى نجد ذلك العنصر !

تخيّل قائمة متسلسلة تحتوي معلومات حول الطلّاب : الاسم، العُمر و المعدّل. سيتم تمثيل كلّ طالب بهيكل نسميه `Student`.

> م
>
> عملنا سابقاً على القوائم المتسلسلة التي تحتوي على `int`. كما قلتُ لك، من الممكن تخزين أي شيء نريد في قائمة، حتى مؤشّراً نحو هيكل آخر كما سأقترحه لك الآن.

إذا أردتُ الوصول إلى المعلومات الخاصة بالشخص Luc Doncieux في الصورة الموالية، يجب عليّ التقدّم في كلّ القائمة كي أكتشف بأنه العنصر الأخير فيها !



بالفعل، لو أننا بحثنا عن الشخص Julien Lefebvre، كان البحث ليكون أسرع بما أنه متواجد في بداية القائمة. و مع ذلك، لتقييم كفاءة الخوارزمية، يجب أن نفكّر دائماً في أسوء الحالات. و الأسوء هو Luc هنا.
هنا، نقول أن خوارزمية البحث لها تعقيد (complexity) *O(n)*، لأنه يجب تصفّح كل القائمة المتسلسلة للوصول إلى العُنصر المراد، و في أسوء الحالات يكون هذا هو آخر عنصر. إذا كانت القائمة تحتوي على 9 عناصر، يجب أن يتم تشغيل 9 دورات للحلقة كحد أقصى لإيجاد العنصر.

في هذا المثال، لا تحتوي القائمة المتسلسلة سوى على أربعة عناصر. سيجد الحاسوب الشخص Luc Doncieux بسرعة كبيرة لا تسمح لنا حتى بأن نقول كلمة "أووه". لكن تخيّل أن هذا الشخص يتواجد في آخر قائمة متسلسلة من 10000 عنصر ! ليس مقبولا أن يتم البحث في 10000 عنصر لإيجاد المعلومة. هنا تتدخّل جداول التجزئة.

2.31 ماهي جداول التجزئة ؟

إذا كنت تتذكّر جيداً، لا تعرف الجداول هذا النوع من المشاكل. لهذا، كي نصل إلى العنصر في الوضعية 2 من الجدول تكفيني كتابة التالي :

```
int table[4] = {12, 7, 14, 33};
printf("%d", table[2]);
```

لو نعطي للحاسوب `table[2]`، فسيتوجّه مباشرة إلى المكان في الذاكرة أين هو مخزّن العدد 14. أي أنه لن يتقدّم في الجدول خانة بخانة.

هل أنت بصدد القول أن الجداول ليست "بذلك القدر من السوء" ؟ لكن في هذه الحالة سنخسر الميزات التي توفّرها القوائم المتسلسلة التي تسمح لنا بإضافة و إزالة خانات في أي لحظة !

في الواقع، القوائم المتسلسلة مرنة أكثر. أما بالنسبة للجداول، فهي تسمح بالوصول السريع للمعطيات. يمكننا القول أن **جداول التجزئة** تشكّل حلّا وسطا بين الإثنين.

يوجد عيب في استعمال الجداول لم نتكلّم عنه سابقاً : يتم تعريف خانات الجدول عن طريق أرقام نسمّيها **الفهارس** (Indices). لا يمكن أن نطلب من الحاسوب : "ماهي المعلومات المتواجدة في الخانة التي تسمّى "Luc Doncieux". أي أننا لإيجاد العُمر و المعدّل لن نتمكّن من كتابة :

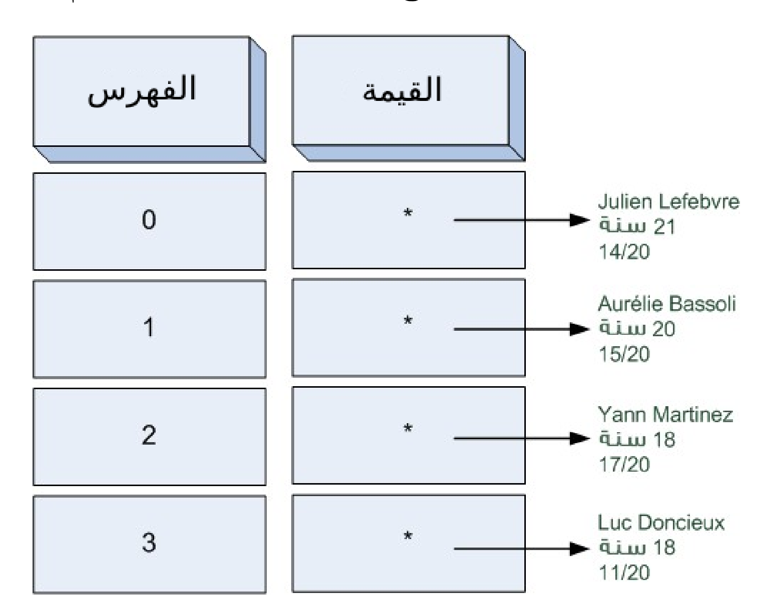
```
table["Luc Doncieux"];
```

مع أنه سيكون عملياً لو أننا نستطيع الوصول إلى خانة ما باستعمال الاسم فقط ! حسناً، هذا ممكن باستعمال جداول التجزئة.

> كما رأينا مؤخّراً. لا تشكّل جداول التجزئة "جزءً" من لغة الـC. نتحدّث هنا عن مبدأ. سنعيد استعمال أساسيات لغة الـC التي نعرفها من قبل لأجل إنشاء نظام ذكي جديد. و كأنه في لغة الـC، باستعمال بعض الأدوات القاعدية، يمكننا إنشاء الكثير من الأشياء !

> بما أنه من الواجب أن يتم ترقيم الجدول بالفهارس، كيف سنجد رقم الخانة لو أننا نعرف الاسم "Luc Doncieux" فقط ؟

ملاحظة جيدة. في الواقع، يبقى الجدول جدولاً و لن يعمل إلا بالفهارس المرقّمة. تخيّل جدولاً يوافق الصورة الموالية : كل خانة لها فهرس و تتوفر على مؤشّر نحو هيكل من نوع `Student`. أنت تعرف القيام بهذا الآن :

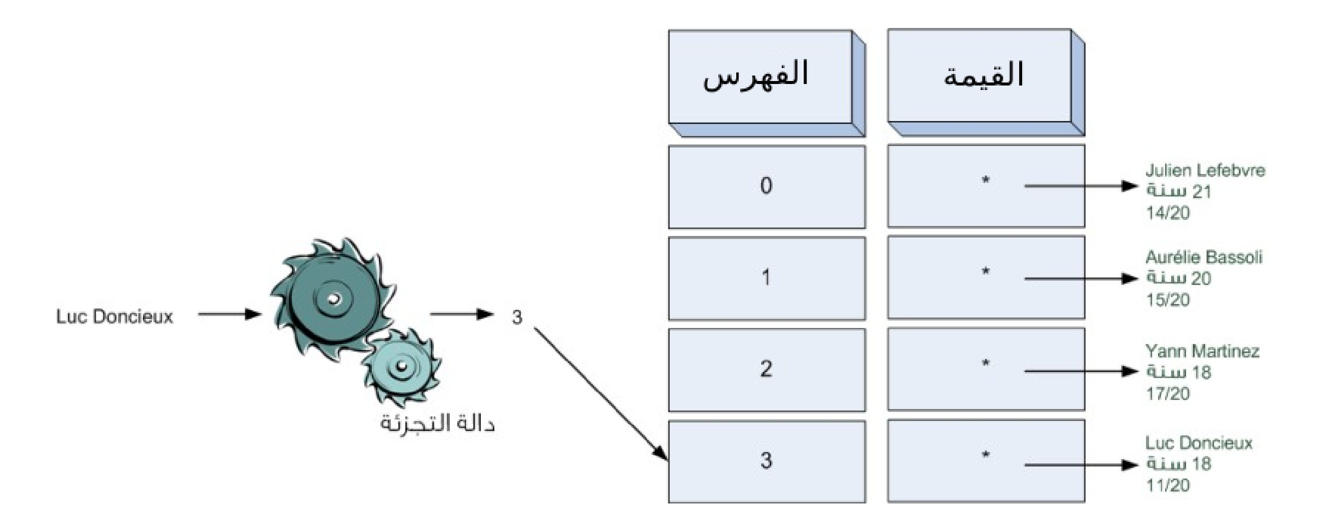


لو أردنا إيجاد الخانة التي توافق Luc Doncieux، يجب أن نجيد تحويل الإسم إلى فهرس في الجدول. و بهذا، يجب أن نتمكّن من ربط كلّ اسم برقم من خانة في الجدول :

- Julien Lefebvre = 0.
- Aurélie Bassoli = 1.
- Yann Martinez = 2.
- Luc Doncieux = 3.

لا يمكننا أن نكتب `table["Luc Doncieux"]` كما فعلتُ سابقاً. لأن هذا غير مسموح به في لغة الـC.

السؤال الذي يُطرح هو : كيف نحوّل سلسلة محارف إلى عدد ؟ هذا هو سحر التجزئة. يجب كتابة دالة تأخذ كمعامل سلسلة محارف، تطبّق حسابات عليها، ثم تُرجع لنا عدداً يوافق تلك السلسلة. سيكون هذا العدد هو فهرس الخانة في الجدول :



3.31 كتابة دالة تجزئة

تكمن كلّ الصعوبة في كتابة دالة تجزئة صحيحة. كيف نحوّل سلسلةً محرفيّة إلى عدد وحيد ؟

أولا و قبل كلّ شيء، لنوضح الأمور : جدول التجزئة لا يحتوي 4 خانات كما أضع في الأمثلة، لكن 100 أو 1000 أو أكثر. لا يهم حجم الجدول، لأن البحث سيكون سريعاً جداً دائما.

> نقول أن هذا تعقيد بدرجة *O(1)* لأننا نجد مباشرة عنصر البحث. في الواقع، دالة التجزئة سترجع لنا فهرسا : يكفي "القفز" مباشرة إلى الخانة الموافقة للجدول. لسنا بحاجة إلى تصفّح كلّ الخانات !

تخيّل إذاً جدولاً من 100 خانة، تقوم فيه بتخزين مؤشّرات نحو هيا كل من نوع `Student`.

```
Student* table[100];
```

يجب علينا أن نكتب دالة، انطلاقاً من اسم، تولّد عدداً محصوراً بين 0 و 99 (رُتب الجدول). هنا يتطلّب منا الأمر الحذاقة. توجد طُرق رياضية جدّ معقّدة كي "نجزّء" البيانات، أي أن نحوّلها إلى أعداد.

> الخوارزميتان MD5 و SHA1 هما دالتا تجزئة مشهورتان، لكنهما متقدّمتان كثيرا بالنسبة لنا حالياً.

يمكنك اختراع دالة التجزئة الخاصة بك. هنا، لكي نبسّط الأمور، أقترح عليك ببساطة أن تجمع القيم ASCII لكلّ حرف من الاسم، أي من أجل الاسم Luc Doncieux ستكون لدينا عملية الجمع التالية :

```
'L' + 'u' + 'c' + ' ' + 'D' + 'o' + 'n' + 'c' + 'i' + 'e' + 'u' + 'x'
```

سيكون لدينا مشكل : هذا المجموع يتخطّى الـ100 ! بما أن الجدول الذي أنشأناه لا يحتوي سوى على 100 خانة، فإن أخذنا بهذه القيمة فسنخاطر بالخروج من حدود الجدول.
أذكّرك بأن كلّ محرف في جدول ASCII يمكن أن يكون مرقّما حتّى 255. و بهذا سنتجاوز بسرعة حاجز الـ100.

لحلّ هذا المشكل، يمكننا استعمال عامل الترديد `%`. هل تتذكره ؟ إنّه يعطي باقي القسمة ! لو نقوم بهذا الحساب :

```
lettersSum % 100
```

سنتحصّل قطعاً على عدد محصور بين 0 و 99. مثلاً، لو أن المجموع يساوي 4315، باقي القسمة على 100 هو 15.
سترُجع إذا دالة التجزئة القيمة 15.

إليك ما يمكن أن تكون عليه الدالة :

```c
int hash(char *string)
{
        int i = 0, hashNumber= 0;
        for (i = 0 ; string[i] != '\0' ; i++)
        {
                hashNumber += string[i];
        }
        hashNumber %= 100;
        return hashNumber;
}
```

لو نعطيها `hash("Luc Doncieux")`، سترُجع لنا القيمة 55. و بـ `hash("Yann Martinez")`، تتحصّل على 80.

بفضل دالة التجزئة هذه، يمكنك أن تعرف في أي خانة من الجدول يجب أن تضع المعلومات ! إذا أردت الوصول إلى هذه الخانات لاحقاً لاسترجاع المعلومة، تكفي "تجزئة" اسم الشخص من جديد لكي نجد فهرس الخانة في الجدول أين تخزّن المعلومات !

أنصحك بإنشاء دالة بحث تتكفّل بتجزئة المفتاح (الاسم) و ترُجع لنا مؤشّراً نحو المعلومات التي نبحثُ عنها. هذا سيعطينا مثلا :

```c
infoAboutLuc = findHashTable(table, "Luc Doncieux");
```

4.31 معالجة التصادمات (Collisions management)

حينما ترُجع دالة التجزئة نفس العدد من أجل مفتاحين مختلفين، نقول أنّه حدث **تصادم**. مثلا في حالتنا، لو أننا نملك شخصاً اسمه تحريك أحرف لـLuc Doncieux، مثلاً Luc Doncueix، سيكون مجموع الأحرف هو نفسه، و بهذا فإن نتيجة دالة التجزئة ستكون نفسها !

يمكن لسببين أن يشرحا التصادم :

- دالة التجزئة لا تعمل بكفاءة عالية. هذا يمثّل حالتنا. لقد كتبنا دالة سهلة جداً (لكن نوعاً ما كافية) من أجل الأمثلة. الدالتان MD5 و SHA1 المذكورتان أعلاه هما ذات جودة عالية لأنهما تنتجان نسبة قليلة من التصادُمات. و لتعلم أن SHA1 مفضّلة في أيامنا هذه أكثر من MD5 لأنها تنتج نسبة تصادمات أقل مقارنة بنظيرتها.

• الجدول الذي نخزن به المعلومات صغير الحجم كثيراً. لو أننا ننشئ جدولا من 4 خانات و نريد تخزين 5 أشخاص، فسيحدث تصادم بالتأكيد، أي ان دالة التجزئة ستُعطي نفس الفهرس من أجل اسمين مختلفين.

إذا حصل تصادم فلا داعي للخوف ! هناك حلان يمكنك الاختيار بينهما : العَنوَنَة المفتوحة و السَلْسَلَة.

العنونة المفتوحة (Open addressing)

إذا بقيت أمكنة شاغرة في الجدول، يمكنك تطبيق التقنية التي تُدعى **التجزئة الخطيّة.** المبدأ سهل. هل الخانة محجوزة ؟ لا يوجد مشكل، سننتقل للخانة التي تليها. آه، هل هذه محجوزة أيضاً ؟ توجّه للتي بعدها.

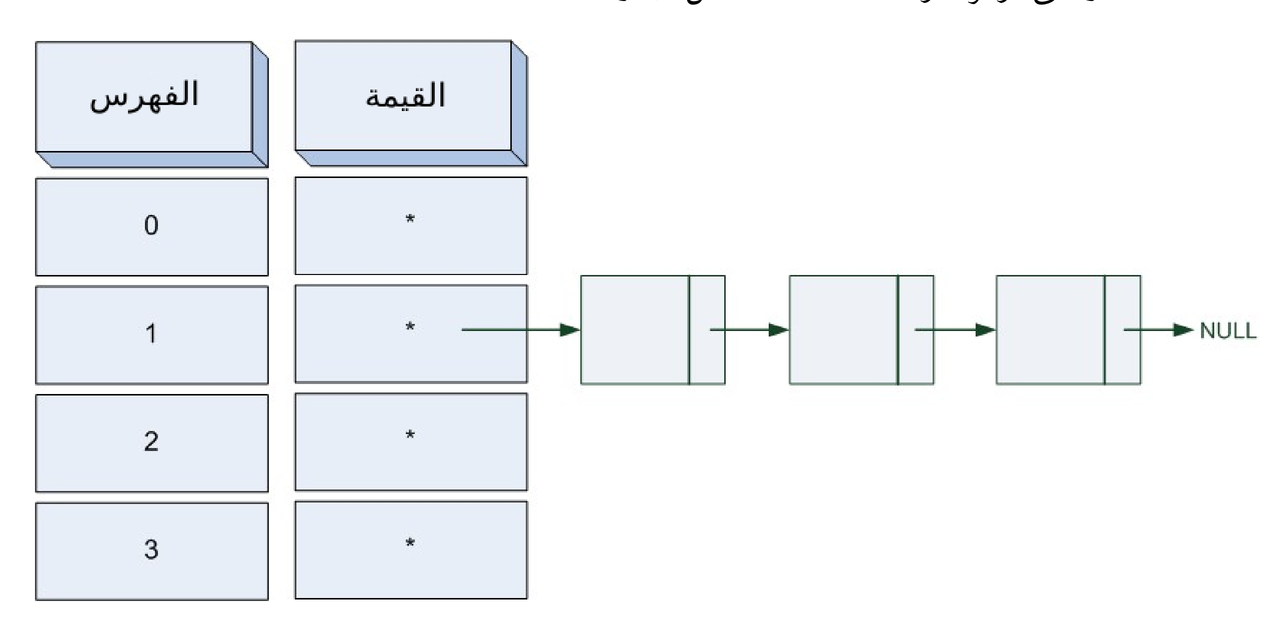
و هكذا حتى تجد خانة موالية فارغة. إن وصلت إلى نهاية الجدول، فعد إلى البداية و أكمل البحث.

تطبيق هذه الطريقة سهل جداً، لكن إن واجهت الكثير من التصادمات، فسيكون عليك استغراق وقت كبير في البحث عن الخانة الشاغرة الموالية.

توجد طرق بديلة (التجزئة المزدوجة، التجزئة الرباعية ...) و التي تنصّ على التجزّئة من جديد حسب دالة أخرى في حالة وجود تصادم. هذه الدوال أكثر كفاءة لكن أكثر تعقيدا من ناحية التطبيق.

السَلْسَلَة (Chaining)

حلّ آخر ينصّ على إنشاء قائمة متسلسلة في مكان التصادم. هل تريد تخزين بيانتين (أو أكثر) في نفس الخانة ؟ استعمل قائمة متسلسلة و أنشئ مؤشّراً نحو هذه القائمة انطلاقاً من الجدول :



بالفعل، سنعود لمشكل القوائم المتسلسلة : إذا كان هناك 300 عنصرٍ في هذا الموقع من الجدول، يجب تصفّح القائمة المتسلسلة إلى حين إيجاد العنصر الصحيح.

هنا، كما ترى. ليست القوائم المتسلسلة دائماً الأمثل، لكن لجداول التجزئة حدودها أيضاً. يمكننا المزج بين الاثنين من أجل الحصول على الجانب الأفضل من كلّ بنية.

على أية حال، النقطة الحساسة في جداول التجزئة هي دالة التجزئة. فكلّما أنتجت تصادُمات أقل. كلما كان ذلك أفضل. سأترك لك مهمة إيجاد دالة التجزئة المناسبة لحالتنا !

ملخّص

- القوائم المتسلسلة مرنة، لكن عملية إيجاد عنصر محدد تستغرق وقتاً طويلاً لأنه يجب تصفّح القائمة عنصراً بعنصر.
- جداول التجزئة هي جداول نخزّن فيها المعلومات في مكان محدد بواسطة دالة التجزئة.
- تأخذ دالة التجزئة مفتاحاً كمعامل (مثلاً : سلسلة محرفيّة) و تعيد عددا كمخرج.
- يتم استعمال هذا العدد لمعرفة عند أي فهرس من الجدول يجب تخزين البيانات.
- دالة التجزئة الأكثر كفاءة هي التي لا تولّد عدداً كبيراً من التصادُمات، أي أنها تجنب قدر المستطاع إرجاع نفس العدد من أجل مفتاحين مختلفين.
- في حالة التصادم، يمكننا استعمال تقنية العنونة المفتوحة (البحث عن خانة شاغرة أخرى في الجدول) أو استعمال تقنية السلسلة (الدمج مع القوائم المتسلسلة).

خاتمة

هل تريد المزيد ؟

لماذا لا نتعلّم لغة الـC++ ؟

http://www.siteduzero.com/tuto-3-5395-0-apprenez-a-programmer-en-c.html

هذا درس آخر كتبتُه حول هذه اللغة قريبة الـC. إذا كنت تعرف الـC، فلن تكون ضائعاً بل ستفهم بسرعة فائقة الفصول الأولى !
فليكن في علمك أنني كتبت درساً قصيرا يسمّى "من الـC إلى الـC++" الذي يبيّن جزءً من الاختلاف بين الـC و الـC++.

http://www.siteduzero.com/tutoriel-3-430167-du-c-au-c.html

بلغة الـC++، يمكنك البدء في البرمجة غرضية التوجّه (أو البرمجة الكائنية) (OOP). قد يكون هذا المبدأ معقّدا قليلا في البداية، لكن ستجد بأن هذه الطريقة في البرمجة ناجعة جداً ! ستكتشف أيضاً معها المكتبة Qt التي تسمح بإنجاز واجهات رسومية كاملة جدّا.

أشكر كثيرا Taurre و Pouet_forever لمساعدتهم الكبيرة في القيام بالمراجعات الأخيرة لهذه الدروس.

ملخّص

هذا الكتاب هو ترجمة لدرس تعلّم البرمجة بلغة الـ C الخاص بموقع OpenClassrooms من الفرنسيّة إلى العربية. يمتاز هذا الدرس بكونه سهل الفهم على المبتدئين في مجال البرمجة، إذ أنّه لا يفترض وجود أيّة مكتسبات قبليّة في هذا المجال لدى القارئ.
يحتوي الكتاب على معلومات مفصلة بخصوص البرمجة بشكل عام و لغة الـ C بشكل خاص، مدعّمة بكثير من المخطّطات التوضيحيّة، الأمثلة و التمارين المصححة.
يتكوّن الكتاب من 31 فصلا موزّعة على 4 أجزاء. فصول الجزء الأوّل تحوّل القارئ من شخص لا علاقة له بالبرمجة إلى مبرمج مبتدئ يجيد التعامل مع الأدوات اللازمة ويجيد المفاهيم الأساسية كأنواع البيانات، المتغيّرات، الشروط، الحلقات التكراريّة و الدوال. في الجزء الثاني، يتعلّم القارئ مفاهيم أكثر تقدّما في البرمجة، كالمؤشّرات، الجداول، السلاسل المحرفيّة والهياكل بالإضافة إلى الحجز الحيّ للذاكرة والتعامل مع الملفّات
بعد ذلك، يتناول الجزء الثالث مبدأ استخدام المكتبات البرمجيّة مركّزا على استغلال مكتبة SDL لإنشاء النوافذ و الرسم فيها و مكتبة FMOD لتشغيل الصوت و استعمالهما لإنشاء برامج و ألعاب حقيقيّة. في الأخير، فصول الجزء الرابع تتناول مواضيع مكمّلة في لغة الـ C تمثّل في تقنيّات الشائعة الاستخدام لتخزين البيانات، كالقوائم المتسلسلة، جداول التجزئة، المكدّسات و الطوابير.

لقد حرصنا على نقل كلّ الأفكار التي قدّمها الكاتب في الدرس الأصلي من بدايته إلى نهايته. كما بذلنا جهدنا في أن يكون النص بسيطا قدر الإمكان و مفهوما للقارئ العربي العادي.

وفي النهاية، لا يسعنا سوى أن نتطلّع بلهفة إلى تجريب برنامجك الخارق الّذي ستنشئه بعد ختامك لهذا الكتاب.

الكاتب الأصلي للكتاب الفرنسي



Mathieu Nebra

مقاول بدوام كامل، كاتب بدوام كامل و مؤسّس مشارك لموقع OpenClassrooms

ترجمة



عدن بلواضح

باحثة في تقنيات الذكاء الإصطناعي و الأنظمة المعلوماتية الذكية، مطورة ألعاب فيديو و كاتبة لعدّة مقالات و مواضيع متعلّقة بالتكنلوجيا و المعلوماتية بعدّة لغات.
eden.belouadah@yahoo.fr

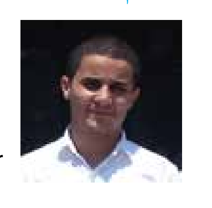
مراجعة و إعداد



حمزة عبَاد

مبرمج متقدّم، مختصّ في التعلّم الآلي و علم البيانات، مدير لأنظمة GNU/Linux ،يملك خبرة صغيرة في تعديل الصور و معرفة بسيطة في التصميم، و هو هاوي إلكترونيات أيضا !
hamza.abbad@gmail.com

تصميم الغلاف



أحمد زبوشي

مبرمج و إختصاصي في الذكاء الإصطناعي، عصامي في هندسة صوتيات و مخطط المعلومات الرسومي، مهتم بالعمل الحر و المعلوماتيات.
zebouchi@live.fr

عدن، حمزة و أحمد متخرجون من جامعة هواري بومدين للعلوم و التكنولوجيا بالجزائر، اختصاص «الأنظمة المعلوماتية الذكية».